SoftSIMD - Exploiting Subword Parallelism Using Source Code Transformations

Stefan Kraemer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr
Institute for Integrated Signal Processing Systems
RWTH Aachen University, Germany
{kraemer, leupers}@iss.rwth-aachen.de

Abstract

SIMD instructions are used to speed up multimedia applications in high performance embedded computing. Vendors often use proprietary platforms which are incompatible with others. Therefore, porting software is a very complex and time consuming task. Moreover, lots of existing embedded processors do not have SIMD extensions at all. But they do provide a wide data path which is 32-bit or wider. Usually, multimedia applications work on short data types of 8 or 16-bit. Thus, only the lower bits of the data path are used and therefore only a fraction of the available computing power is exploited for such algorithms. This paper discusses the possibility to make use of the upper bits of the data path by emulating true SIMD instructions. These instructions are implemented purely in software using a high level language such as C. Therefore, the application can be modified by making use of source code transformations which are inherently portable. The benefit of this approach is that the computing resources are used more efficiently without compromising the portability of the code. Experiments have shown that a significant speedup can be obtained by this approach.

1 Introduction

The vast majority of processors shipped today are embedded processors, due to the increasing demand for consumer electronics. Such processors are often used in digital signal processing and media processing applications for which performance is critical. Therefore, hand written assembly code is used to obtain maximum performance. However, the growing complexity of embedded systems has led to higher costs for algorithm and software development. Today, the costs of algorithm and software development are dominating the development of an embedded system. Hence, in recent times a move towards programming embedded devices in high level languages (HLL) such as C can be observed. Such HLLs enable a faster software development process and increase the portability of the code. Furthermore, a lot of legacy code is available in C which can be reused, thereby reducing the development and verification effort. The catch, however, is that the usage of a HLL comes at the price of a less efficient implementation than hand written assembly code[6].

In order to reduce the negative impact of HLL on performance, developers are using compiler known functions and assembly for performance critical parts of the application. In a different approach, designers rely on rewriting the HLL in such a way that the compiler can generate code with higher quality. The advantage of such an approach is that source code transformations are inherently portable and can result in a significant performance improvement[7].

Multimedia applications contain a lot of inherent parallelism which can easily be exploited by using SIMD instructions. With very limited hardware overhead it is possible to boost the application performance significantly.

Many processor manufacturers have developed their own SIMD extension, like Intel (MMX), Motorola (AltiVec) and AMD (3DNow!). Recently, also many embedded processor manufacturers (TI C6x, Analog Devices ADSP 21xxx) have adopted SIMD instructions, due to the fact that multimedia applications have become more and more important for the high performance embedded processor domain. The major disadvantage is that the different SIMD implementations are incompatible with each other. Each implementation provides only a small number of SIMD instructions and a set of pack and unpack instructions which are very special to the target architecture. Consequently, the compiler support for SIMD instructions is limited. In order to obtain good results software developers have to rely on compiler intrinsics to manually improve the code.

This paper focuses on the potential of implementing SIMD (*Single Instruction Multiple Data*) instructions purely in software. We call this kind of implementation *SoftSIMD*. By using this software centric approach it is possible to easily integrate SIMD instructions in HLLs. Furthermore, the implementation is completely independent of the target architecture. Hence, the major advantage of HLLs

978-3-9810801-2-4/DATE07 © 2007 EDAA

- the portability - is not compromised. In addition, SoftSIMD instructions do not require any special hardware support. They can be applied to nearly any target architecture. Of course, the speedup that can be achieved is lower than what can be obtained with hardware support.

In an article recently published on DSPDesignLine[1] different possibilities how to operate on several data elements at the same time are discussed. This emphasizes the necessity of being able to handle data in parallel without hardware support.

The rest of the paper is organized as follows. After a discussion of related work in Section 2, the possibilities to implement SIMD instructions using a pure software implementation are discussed in Section 3. Furthermore, the prerequisites and limitations are explained in Section 4. The results obtained with the SoftSIMD approach are presented in Section 5. Section 6 concludes this paper and gives an outlook on future work.

2 Related Work

In order to effectively exploit SIMD instructions it is essential that the compiler is aware of this type of special instructions. Otherwise, the user has to rely on hand optimized libraries to make efficient use of the SIMD capabilities of the processor. Therefore, the research focuses on effective compiler support for all kinds of SIMD instructions.

Many of todays compilers provide a semiautomatic SIMD support. The programmer uses *compiler known functions (CKF)* or *intrinsics* to steer the SIMDfication process. The CKFs act as placeholders for SIMD instructions. During the compilation process they are replaced by the appropriate SIMD assembly instructions. However, due to the low-level programming style and the poor portability of code with CKFs, this cannot be considered a satisfactory solution. Some advanced compilers (e.g., for Intel MMX and SSE) provide automatic generation of SIMD instructions, yet restricted to certain C language constructs. Moreover, these compilers are inherently non-portable.

In the domain of "general purpose" retargetable compilers, recent versions (4.x) of the *gcc*[11] support SIMD for certain loop constructs, but *gcc* is generally known as being difficult to adapt efficiently to embedded processor architectures.

In research on embedded processor code optimization, a number of techniques for SIMD utilization have been proposed recently. In [10] a combination of traditional code selection[5] and Integer Linear Programming based optimization is presented. This approach achieves high code quality but suffers from high complexity for large programs. The work in [9] presents an efficient approach for packing operations step by step into SIMD instructions, and it presents results for the AltiVec ISA. Further works in this domain deal with memory alignment optimization for SIMD[3], pointer alignment analysis[12], and flow graph permutations for more effective SIMD instruction packing[8].

Closely related to the work presented in this paper is the SWAR technique[4]. Information how to manipulate multibyte data can be found in [13].

In summary, a number of compiler techniques for using SIMD instructions are available. Some of these techniques are tailored for special architectures, others can be retargeted. All approaches require special hardware support as well as compiler support. This paper, however, emphasizes the possibility to emulate SIMD instructions without dedicated hardware. The SoftSIMD instructions can be obtained by applying source code transformations to the code. Hence, this approach is also independent of the underlying compiler.

3 SoftSIMD: Exploiting Subword Level Parallelism

In image processing, mostly 8-bit and 16-bit data are processed on 32-bit processors. Hence, a significant part of the data path remains unused. By using the SoftSIMD approach it is possible to increase the average utilization of the data path leading to an increased performance.

3.1 SoftSIMD Instructions

To understand why special hardware support is required for SIMD instructions we will have a closer look at the implementation of a SIMD instruction, for example a SIMD 4 addition. This operation performs four 8-bit additions in parallel using a 32-bit adder. Each subword addition must be independent of all the other additions, therefore it has to be guaranteed that the carry path ends at the most significant bit of each subword. This can be achieved by inserting three AND gates in the carry path of the 32-bit adder as shown in Figure 1. Depending on the signal applied to the AND gates, the adder acts as a regular 32-bit adder or a SIMD 4 adder. As can be seen from this example, only minor hardware modifications are required to make sure that each data element can be processed separately. The low hardware overhead and the potentially high speedups are the reasons why SIMD operations are so popular for improving the performance of applications.

In order to obtain the same behavior with a pure software implementation it has to be guaranteed that the carry path between different subwords is interrupted. This can be achieved by introducing guard intervals between each subword. The simplest guard interval is a zero bit which is introduced between each subword. Naturally, using guard

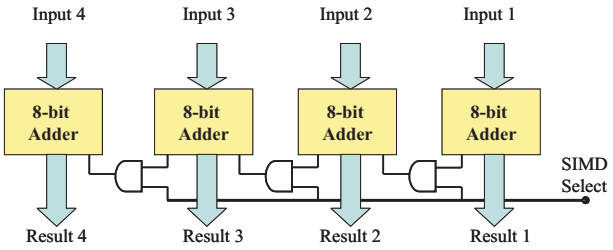

Figure 1. Modified 32-bit adder, depending on "SIMD select" it acts as a SIMD adder or a standard 32-bit adder

intervals reduces the number of bits that can be used for computation. In case of a 32-bit adder it is only possible to implement a SIMD 2 addition or a SIMD 3 addition operating on 8-bit data types. The remaining 8-bits are used to form the guard bits.

Using a zero bit as guard bit is only possible in case that all operands are positive. If one or more operands are negative or a subtract operation is performed then a different type of guard bits is required. The reason for this is the way negative values are represented: the two's complement. A negative value is transformed into the two's complement by bitwise invertion of the absolute value and then adding a one. Hence, all unused upper bits are set to one (sign extension). In order to cope with this, two different approaches are possible:

- Use modified guard bits
- Operate only on positive values

In the first approach the guard bits are modified to make sure that the carry path is interrupted also in case of a subtraction. As described in [4] the guard bits of the subtrahend need to be set to one and the guards bits of the subtractor are set to zero. The following example shows a SIMD 2 subtraction performed fully in software:

```
result=( (a|0x80008000) -b) & 0x7FFF7FFF
```

As can be seen in this example, two additional operations, `|` and `&`, are required to perform the SIMD subtraction. Since the overhead caused by the `&` and the `|` operation needs to be compensated, a speedup can only be obtained if the parallelism is equal to or higher than 3.

The second approach is to make sure that all input and output values are always positive, especially the result of a subtraction. This can be achieved by adding a constant to each variable. The constant must be greater than or equal to the smallest absolute value that can be represented by the variable. Here, A denotes a variable which holds a value which can range between $-M_{Max}$ and M_{Max}.

$$\begin{aligned} A &\in \{-M_{max}, ..., M_{max}\} \\ A_{bias} &= A + M_{max} \\ A_{bias} &\in \{0, ..., 2\,M_{max}\} \end{aligned}$$

By adding the bias M_{Max} to the variable A the resulting variable A_{bias} only holds positive values. Thus, the standard SoftSIMD approach for additions can be applied. However, if more consecutive additions or subtractions occur in the application, then the user must make sure that all intermediate results as well as all final results will produce only positive solutions. Hence, it is not possible to use the same bias for all input variables. The bias for each input must be chosen in such a way that the output will always produce a positive value. In the following variables A and B store a value which is in the interval of $-M_{Max}$ to M_{Max}. Figure 2 shows a subtraction of two variables us-

$$\begin{aligned} A, B &\in \{-M_{max}, ..., M_{max}\} \\ A_{bias} &= A + M_{max} \\ B_{bias} &= B + M_{max} \\ C &= A_{bias} - B_{bias} = A - B \\ C &\in \{-2\,M_{max}, ..., 2\,M_{max}\} \end{aligned}$$

Figure 2. By using only one offset it is not guaranteed that the result is always positive

$$\begin{aligned} A, B &\in \{-M_{max}, ..., M_{max}\} \\ A_{bias} &= A + 3\,M_{max} \\ B_{bias} &= B + M_{max} \\ C &= A_{bias} - B_{bias} = A - B + 2\,M_{max} \\ C &\in \{0, ..., 4\,M_{max}\} \end{aligned}$$

Figure 3. By using two different offsets the obtained result will always be positive

ing the same bias. The result C of the subtraction, however, is not guaranteed to be positive since the bias of both variables compensates each other. The variable C holds a value between $-2\,M_{Max}$ if both input variables are $-M_{Max}$ and $2\,M_{Max}$ if both inputs are M_{Max}. Figure 3 exemplifies the same subtraction, but this time both variables are shifted using two different constants. As shown in Figure 2 the smallest value obtained in a subtraction is $-2\,M_{Max}$. To guarantee always a positive result for the subtraction the bias must be at least $2\,M_{Max}$. In order to obtain this bias, variable A uses $3\,M_{Max}$ as bias and variable B uses M_{Max} as bias. Thus, the resulting bias for variable C is $2\,M_{Max}$. Therefore, it can be guaranteed that the output of the subtraction will always produce a positive result.

For this kind of approach it is crucial to choose the correct offset M_{Max} for each input variable. Thus, additional knowledge about the range of the input and output variables proves to be useful in order to select adequate offsets. The overhead introduced by this SoftSIMD operation is three extra operations for adding and removing the bias. But in contrast to the first approach the number of overhead instructions depends only on the number of inputs and out-

puts of the SIMD block. Figure 4 depicts a code example

```
1 :  unsigned char g0,g0x,g1,g1x,h0,h0x,h1,h1x;
2 :  signed short r1,r1x
3 :  ...
4 :  /* Add bias*/
5 :  g1 = g1 + 255;  g1x = g1x + 255;
6 :
7 :  g0_simd = pack2in32(g0,g0x);
8 :  g1_simd = pack2in32(g1,g1x);
9 :  h0_simd = pack2in32(h0,h0x);
10:  h1_simd = pack2in32(h1,h1x);
11:
12:  /*perform simd operation*/
13:  r1_simd = g1_simd - h1_simd + g0_simd + h0_simd;
14:
15:  unpack2in32(r1_simd,&r1,&r1x);
16:
17:  /*Remove bias*/
18:  r1 = r1 - 255;  r1x = r1x - 255;
19:  ...
```

Figure 4. Using a bias to make sure that all results are always positive

which uses a bias to make sure that the computation will always produce a positive result. All input variables are of type `unsigned char`. In the example one SIMD subtraction is performed, in order to guarantee a positive result a bias of 255 is added to the variables `g1` and `g1x`. After the SIMD calculation the variable `r1_simd` is unpacked and the bias is subtracted to obtain the final result.

After understanding how a SIMD addition can be realized with a pure software implementation we will discuss how and under which conditions SIMD multiplications can also be implemented.

In principle the idea of using a guard interval to interrupt the carry path applies also for the SIMD multiplication. However, the multiplication of two variables with k number of bits each produces a result with $2k$ bits. Therefore, the guard interval must be at least k bits wide. Thus, with a 32-bit multiplication it is only possible to implement a SIMD 2 multiplication for two 8-bit values in parallel.

To understand what happens if two variables with two operands each are multiplied, it is necessary to understand how a multiplication works. A binary multiplication can be seen as consecutive addition and shifting of one operand. The number of additions and the number of shift operations is determined by the second operand. In case of a packed variable which is multiplied with a scalar variable p all subwords will be multiplied independently by p. Hence, it is possible to model a SIMD multiplication in case that both subwords will be multiplied with the same factor. Note, that this type of multiplication is a special case ($x = 0$) of the dot product explained in the following.

In multimedia applications most of the time is spent in small loop kernels which are ideal candidates for SIMD instructions. By unrolling a loop the parallelism becomes explicit and the loop can be SIMDfied. For SoftSIMD, our experiments have shown that unrolling the outer loop instead of the inner loop of the kernel produces the required SIMD structure where several subwords are multiplied with the same factor. Figure 5 shows part of a convolution with a 3×3 filtering mask. By unrolling the outer loop by a factor of two the input variables `c1[i+j*k]` and `c1[i+(j+1)*k]` are multiplied with the same factor `mask[i]`. Figure 6 shows the unrolled code. In general

```
1 :    for (j = 0; j < 256 ; j++){
2 :       ...
3 :       for (i = 0; i < 3; i++){
4 :           s0 = c1[i+j*k] * mask[i];
5 :           ...
6 :         }
7 :       ...
8 :     }
9 :
```

Figure 5. Convolution

```
1 :    for (j = 0; j < 256 ; j+=2){
2 :       ...
3 :       for (i = 0; i < 3; i++){
4 :           s0 = c1[i+j*k] * mask[i];
5 :           s0_2 = c1[i+(j+1)*k] * mask[i];
6 :           ...
7 :         }
8 :       ...
9 :     }
```

Figure 6. Unrolled Convolution

it is not possible to model SIMD multiplication using SoftSIMD. However, the multiplication of two packed operands results in a complex SIMD instruction. To understand this behavior let 2^d denote a displacement of d-bits. Thus, the two input operands can be written as:

$$\begin{aligned} OperandA &= a \cdot 2^d + b \quad a, b \in \{0, ..., M_{Max}\} \\ OperandB &= x \cdot 2^d + y \quad x, y \in \{0, ..., M_{Max}\} \end{aligned}$$

The multiplication of both operands produces the following result:

$$OperandA \cdot OperandB = ax \cdot 2^{2\,d} + (ay + bx) \cdot 2^d + by$$

In case of a 32-bit multiplication with an displacement of $d = 16$ bits only the results with a displacement smaller than $2\,d = 32$ bits are visible. The result contains in its upper half a dot product of the four input subwords and in the lower half a product of two input subwords. The possibility not to model only basic SIMD instructions but also to be able to model complex SIMD instructions has proven to be very useful, since in standard DSP applications the dot product occurs frequently. It is important to note that the dot product cannot be applied without carefully checking the value ranges of the operators.

If all input variables range between 0 and M_{Max} then the product of two variables results in a range between 0 and

$M_{Max} \cdot M_{Max}$. In case of the dot product a further addition is required which doubles the maximum range. Therefore, the following value ranges are obtained for the upper and the lower part of the multiplication of $OperandA$ and $OperandB$.

$$\begin{aligned} ay + bx &\in \{0, ..., 2\,M_{Max}^2\} \\ by &\in \{0, ..., M_{Max}^2\} \end{aligned}$$

In case of a 8-bit multiplication the result of the dot product is at most 17 bits wide. Hence, it is not possible to represent it with the upper 16 bits. If all operands are only 7 bits wide, the dot product yields a 15-bit result. To effectively use the dot product, additional knowledge about the value range of the input variables is required. Though, there is a possibility to extract the 17th bit from processor by reading the carry flag of the processor. This can only be done be using assembly instructions which would make the complete approach machine dependent. For the sake of portability the input variables need to be limited to 7-bit. The convolution shown in Figure 5 serves as example to demonstrate the complete transformation flow. The first step is to unroll the code to make the parallelism explicit, as shown in Figure 6. Once the code is unrolled the SoftSIMD transformations can be employed. Figure 7 depicts the code example after the SoftSIMD transformations have been applied to it. It is assumed that each array element of `c1` is stored using a 16-bit alignment. Thus, it is possible to load data which is already packed. In case this is not possible, each data element needs to be loaded and afterwards packed into a single variable. The next step is to perform the SIMD multiplication. After the inner loop is completed, the obtained result `s0_simd` needs to be unpacked.

```
1 :  for (j = 0; j < 256 ; j+=2){
2 :    ...
3 :    for (i = 0; i < 3; i++){
4 :      /* SIMD load */
5 :      c1_simd  = *((unsigned int *) (c1 + i+j*k))
6 :      /*SIMD MUL*/
7 :      s0_simd  = c1_simd * mask[i];
8 :      ...
9 :    }
10:    /* unpacking the result */
11:    sum1 = (s0_simd >> 16);
12:    sum2 = s0_simd & 0xFFFF;
13:    ...
14: }
```

Figure 7. Convolution using SoftSIMD

4 SoftSIMD Boundary Conditions

In the previous section it was shown how to mimic the behavior of a SIMD instruction purely in software. But the boundary conditions for the SoftSIMD instructions have not been considered yet. First of all, the application must fulfill all requirements for regular SIMD instructions such as data alignment, parallelism, and correct data bit width to name only the most important requirements. On top on this the implementation of SoftSIMD instructions has additional requirements.

Using just one guard bit as a buffer between two operands has been shown not to be enough in a real application, since the SoftSIMD implementation pays only off if a sequence of operations can be executed on packed data. Therefore, the number of guard bits depends heavily on the number of operations which are executed on packed data, e.g. two subsequent additions. In case that additional information about the value range on the variables is available it is possible to determine the maximum number of required bits per operand. Thus, it is possible to pack the data more efficiently than assuming the worst case bit width for each operand. Furthermore, the value range information is crucial for determining the offset of each input. Without such information in both cases the worst case needs to be assumed, which will reduce the performance of the implementation. Additionally, as described in Section 3.1 working with signed data causes a significant overhead which can be hard to overcome. Therefore the ideal application for SoftSIMD instructions should use only a minimum number of signed data types and work on 8-bit wide data elements. The domain of image processing satisfies all these constraints, each pixel of a image is described as unsigned 8-bit value. In some cases, the 8-bit data is stored with a 16-bit alignment allowing loading pre-packed data directly from the memory with a 32-bit load operation. In addition, the image processing algorithms contain a lot of parallelism which can be exploited by using SIMD instructions.

5 Results

To measure the impact of SoftSIMD instructions a set of 6 different benchmarks from the image processing domain have been chosen. The benchmarks range from fairly simple algorithms like vector addition and dot product to more complex image processing filters like sobel. As target architecture a MIPS32 24K[2] is used with the gcc 3.3.4 C-compiler.

Figure 8 shows how the speedup varies depending on the number of loop iterations. With increasing number of iterations the speedup also increases until it saturates. Since the used kernels are fairly small, the overhead introduced by the loop limits the possible speedup in case of few iterations. For a larger number of iterations the impact of the loop becomes negligible. Then, the maximum achievable speedup with the SoftSIMD implementation is reached. The other three benchmarks have more complex loop bodies, therefore the impact of the loop on the performace is minimal.

Figure 9 displays the obtained speedups for the different

applications. The vector addition is benchmarked with two different degrees of parallelism $\rho = 2$ and $\rho = 3$. All results show a significant speedup which ranges from 1.17 to 2.28. However, in all cases the theoretical limit is not reached, due to overhead introduced by packing and unpacking the data and by handling subtractions. Interestingly, the dot product shows lower speedup than expected. The complex SIMD instruction replaces two multiplications and one addition, thus, the theoretical speedup is 3. However, instead of using two 8-bit multiplications a 32-bit multiplication is used, which has a longer latency on our target architecture, therefore, most of the speedup is lost.

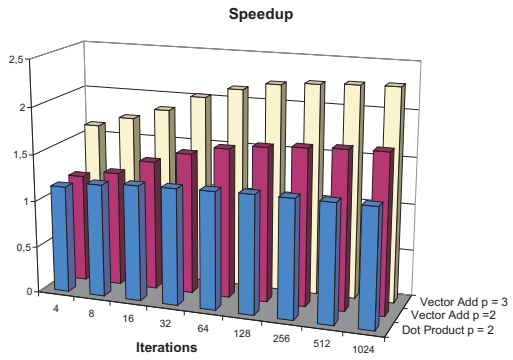

Figure 8. Obtained speedup depending on the number of iterations

Figure 9. Speedup achieved using a pure Software Implementation of SIMD Instructions on MIPS32 24K

6 Conclusion

The results clearly indicate that it is possible to increase the utilization of a wide data path in multimedia applications. Without any hardware modifications it is possible to speed up applications by exploiting the available parallelism. For the image processing domain speedups between 1.17 and 2.28 have been measured. The number of applications that can benefit from this approach is rather limited due to the constraints described earlier. However, quoting[1], "*...those few cases where it's applicable, it can help you meet performance targets you would otherwise miss, and it may be less painful than the alternatives.*"

For the future it might be interesting to automate the SoftSIMD process and integrate it in a compiler. Thus, the compiler would be able to automatically generate a target independent SIMD implementation. Due to the close relation of SIMD and SoftSIMD a lot of existing work could be reused. Additionally, a good value range analysis would be required. Furthermore, it would be interesting to analyse if a mixed approach, where some parts are implemented in hardware and other parts are implemented in software, can improve performance. For example, hardware support for packing and unpacking of data could reduce the overhead.

References

[1] *BDTi: Emulating SIMD in Software*. http://www.dspdesignline.com/showArticle.jhtml?articleID=192501356.

[2] *MIPS32 24K Family*. http://www.mips.com.

[3] A. Eichenberger and K. O. P. Wu. Vectorization for SIMD architectures with alignment constraints. In *Proc. Programming Language Design and Implementation (PLDI)*, 2004.

[4] R. J. Fisher. *General-Purpose SIMD within a Register: Parallel Processing on Consumer Microprocessors*. PhD thesis, Purdue University, 2003.

[5] C. Fraser, D. Hanson, and T. Proebsting. Engineering a Simple, Efficient Code Generator Generator. In *ACM Letters on Programming Languages and Systems, vol. 1, no. 3*, 1992.

[6] A. Frederiksen, R. Christiansen, J. Bier, and P. Koch. An Evaluation of Compiler-Processor Interaction for DSP Applications. In *34th IEEE Asilomar Conference on Signals, Systems, and Computers*, 2000.

[7] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, 2000.

[8] A. Kudriavtsev and P. Kogge. Generation of Permutations for SIMD Processors. In *Proc. Languages, Compilers and Tools for Embedded Systems (LCTES)*, 2005.

[9] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proc. Programming Language Design and Implementation (PLDI)*, 2000.

[10] R. Leupers. Code Selection for Media Processors with SIMD Instructions. In *Design Automation & Test in Europe (DATE)*, 2000.

[11] D. Naishlos. Autovectorization in GCC. In *Proceedings of the GCC Developers' Summit*, pages 105–118, http://www.gccsummit.org/2004, 2004.

[12] I. Pryanishnikov, A. Krall, and et al. Pointer Alignment Analysis for Processors with SIMD Instructions. In *Proc. 5th Workshop on Media and Streaming Processors*, 2003.

[13] H. S. Warren. *Hacker's Delight*. Addison Wesley, 2002.