

Optimization and code parallelization for
processors with multimedia SIMD instructions

François Ferrand

27th August 2003

Abstract

For about a decade, modern processors have been gifted with a new type of instructions, designed for use in multimedia applications. Providing a limited scale SIMD execution model, these instructions can dramatically improve the performance of today's applications, at the expense of development time. For there is indeed little support from the compiler, and it is up to the developer to use those new instructions properly, using assembly language or compiler intrinsics.

This paper presents the work that has been done during my master's thesis at the *Laboratoire d'Informatique des Télécommunications* of the ENST Bretagne to support the automatic generation of code optimized with those multimedia instructions, using the PIPS compilation platform.

Résumé

Depuis une dizaine d'années, les processeurs ont été dotés d'un nouveau type d'instructions, destinées aux applications multimédia. En donnant accès à un mode d'exécution SIMD de petite échelle, elles permettent des gains de performance spectaculaires dans de nombreuses applications courantes, au prix d'un temps de développement accru. En effet, les compilateurs tardent à fournir un réel support de ces instructions, et il est par conséquent nécessaire de recourir à l'assembleur ou à des pseudo-fonctions propres au compilateur (*compiler intrinsics*).

Ce rapport présente le travail réalisé dans le cadre de mon DEA au *Laboratoire d'Informatique des Télécommunications* de l'ENST Bretagne, qui consiste à supporter la génération automatique de code optimisé à l'aide de ces instructions, en utilisant la plate-forme de compilation PIPS.

Table of Contents

Abstract	iii
Résumé	v
Introduction	1
1 Multimedia extensions to general purpose processors	3
1.1 Introduction	3
1.2 SIMD extensions	4
1.3 Instruction set overview	5
1.3.1 Data path considerations	5
1.3.2 Supported data types	6
1.3.3 Supported instructions	6
1.4 Feature comparison	7
1.5 How to use it ?	8
2 Existing Compilation methods for multimedia instruction sets	11
2.1 Vectorization	11
2.2 Superword Level Parallelism	13
2.3 Pattern matching	14
2.4 Code selection	15
3 Machine abstraction	19
3.1 High level SIMD execution model	19
3.2 High level description of the available parallel instructions	20
4 A simple vectorization method	23
4.1 Overview	23
4.2 Atomizer	23
4.3 Optimized unrolling	24
4.4 Reduction detection and elimination	26
4.5 Single assignment form transformation	27
4.6 Statement reordering and grouping	29
4.7 Group vectorization and SIMD vector dependence analysis	31

4.8	SIMD code generation	34
5	SAC: SIMD Architecture Compiler	37
5.1	PIPS	37
5.2	SAC overview	39
5.3	SIMD Atomizer	40
5.4	Reduction Elimination	40
5.5	Single Assignment Form	42
5.6	Simdizer	42
5.7	A full PIPS/SAC script	44
6	Experimental results	47
7	Perspectives	51
	Conclusion	53
	Bibliography	55

Introduction

Recently, a new class of applications making use of rich graphics, sound and videos, has placed new demands on general purpose microprocessors. These processors, based on architectures proposed more than 10 years ago, following the CISC or RISC principles, were designed for the purpose of performing operations on integers or floating-point numbers, and to handle character-based data. They were never supposed to deal with high-demanding applications, like movie playback. As a consequence, it comes as no surprise most of these processors can hardly handle this task.

To reverse this trend, microprocessor vendors have introduced some extensions in their newer generations of processors: MMX on Intel processors, VIS on SUN processors. . . These can greatly increase the performance of media intensive applications, such as graphics and multimedia applications, for which these extensions have been designed. Yet, their actual field of application is much wider, as they are also used in telecommunications, image and signal processing, simulation and scientific calculations. They can be used in almost any application that requires a lot of processing power. As it is integrated in almost all of today's computer systems, this approach provides a cost-effective alternative to task-specific hardware accelerators.

Unfortunately, these instructions do not provide transparent performance improvement. The applications need to be reworked to take advantage of this new technology, to extract the parallelism and to reorganize the data appropriately. Consequently, this method is most of the time reserved to libraries provided by the microprocessor vendor, or to expert programmers. In any case, they are used only to optimize the critical parts of a system. To allow for a wider and faster development of these techniques in more general applications, new compilation techniques needs to be developed, releasing the programmer from the burden of writing the optimized code.

This report presents the work done during my DEA¹ internship, starting with a bibliography which presents the available opportunities offered by this new class of instructions, and the solutions to the problem of automatic generation of optimized code using these extensions.

The first part will thus describe this new architecture and programming

¹*Diplôme d'Études Approfondies*, the French equivalent for the Master of Science degree.

model, also called SWAR² by Randall Fisher [1, 2] or microSIMD [3]. A presentation of the various instruction sets, as well as a comparison of their features, is also included. The second part will then discuss the various algorithms that can be used to generate the optimized code. These include the classical SIMD technique of vectorization, and a variation of this technique which is much simpler yet gives the exact same results. Two other methods are presented, which are more experimental but provide better results by increasing the scope of use of the extensions. The third part presents various problems related to the variety of extensions. Indeed, not all instruction sets are equal, thus some work may be needed to provide a generic optimizer, in order to abstract the actual multimedia extension used.

The next part describe the actual work that I did on SAC, the SIMD Architecture Compiler, a new compiler targeting the multimedia extensions of general purpose processors. This compiler has been implemented as a source-to-source compiler on the PIPS platform. The fourth section thus describes the algorithms and methods used, while the next one goes into more details regarding the implementation on the PIPS platform. Finally, I present some experimental results for this compiler, and give some perspectives on how to make the generated code even more efficient.

²*SIMD Within A Register.*

Chapter 1

Multimedia extensions to general purpose processors

1.1 Introduction

Nowadays, multimedia applications are widely used. As a consequence, the high processing power which used to be needed only for research or military projects is now required at home by millions of users. Not so long ago, the only solution for the user would have been to buy an expensive, specialized, coprocessor. However, with the advances of modern silicon technology, designers are left with lots of space, allowing them to pack much more gates than before. Yet, there is no real need for that much space, as increasing the gates count would certainly result in longer critical path, thus lowering the processor speed. Thus, the typical response has been to increase the cache size. This is particularly true for RISC processors, where the logic is designed to be as simple (and thus as fast) as possible.

As the demand for higher performance for off-the-shelf processors grew, vendors have quickly realized they could improve the performance of those media intensive applications at little cost by using this extra space to implement specific multimedia extensions. These extensions, implementing a minimal set capable of accelerating media-intensive applications, come with the promise of significant gains in multimedia applications.

Thus, during the mid 90s, almost all general-purpose processor vendors have added multimedia extensions to their processors, designed to complement the “regular” instructions and improve the performances in today’s multimedia applications. Here is a list of available instruction sets, and the architecture they extend:

- 3DNow!, on AMD K6 and Athlon processors [4, 5];

- MMX¹, on Intel Pentium II processors [6, 7];
- AltiVec on Motorola Power PC G4 processors [8];
- MAX and MAX-2 on HP PA-RISC and PA-RISC2 processors [9];
- MIPS V and MDMX² on Silicon Graphics processors [10];
- MVI³ on Digital Alpha processors [11];
- SSE⁴ and SSE2, on Intel Pentium III and Pentium 4 processors [7];
- VIS⁵ on SUN Sparc v9 processors [12].

Just as many other technologies, these extensions are not yet used to their full potential, even though a new generation is going to appear soon: Intel has already introduced a new multimedia extension on its Itanium line of processors [13], Motorola is working on the SPE instruction set. . . These new multimedia extensions come with the promise of even better performance, lower power consumption. . .

1.2 SIMD extensions

In multimedia applications, the samples to be processed are usually quite small, yet numerous. As a consequence, 8 or 16 bits are usually enough to handle a single sample: this is the case for example for audio data (16 bits per sample) and images (8 bits per color). On the other hand, modern microprocessors are 32 or 64 bit processors: this means they work with 32 or 64 bit operands. This means that those applications waste an important part of the available processing power.

To improve the performance of those data intensive applications, almost all processor vendors have extended their architecture to add support for a new class of instructions, operating on lower precision operands. As a specific case of the well known SIMD⁶ model, they allow to perform more the same operation on more than one sample at a time. If the processor allows to perform an operation on, say, 8 elements at a time, then one can expect to get a speedup of up to 8 times! Obviously, this theoretical limit is very seldom achieved, but the technology has proved quite useful.

The main idea behind these extensions is that the content of a single register can be logically considered as composed of a few smaller, distinct elements. For

¹MultiMedia Extension.

²MIPS Digital Media Extension, read as “MaDMaX”.

³Motion Video Instructions.

⁴Streaming SIMD Extensions.

⁵Visual Instruction Set.

⁶*Single Instruction flow, Multiple Data flow.*

example, the data contained in a 64 bit register can be seen as a single 64 bit element, but also as 8 8-bit elements. This means these instructions work on small data elements packed together into a single register: the resulting data types are thus called “packed data types”. The fact that all multimedia extensions work this way justifies the name “SWAR”, or “SIMD Within A Register”, given by Randall Fisher to this new class of instructions [1, 2].

This way, the extensions can provide instructions that perform the same operations on all data elements of a register, thus providing a limited form of SIMD for the general purpose processors. This is especially interesting for multimedia applications, where data is usually small: while keeping registers with the same width, it is now possible to perform the computation on more than one element. As the actual elements are small, the waste of register space is now much more limited. This form of parallelism is labeled “subword parallelism” by Hewlett-Packard in its MAX-2 extension [9].

1.3 Instruction set overview

1.3.1 Data path considerations

In addition to this throughput increase, the extensions also include common instructions often found in core loops of multimedia applications, allowing performing them with a latency of one cycle, thus leveraging important performance increase. This is the case, for example, of the multiply-add instruction.

In super-scalar microprocessor, a technique used to increase the intrinsic parallelism of execution is to have separate functional units and data path for integer and floating-points operations. The same concept is used for multimedia extensions. Depending on the vendor choice, the multimedia extension uses the integer data path (MAX-2), the floating-point data path (MMX, MDMX, VIS), or even a separate data path (SSE, SSE2).

Obviously, the most promising option, from the performance point of view, is to have a separate data path, but this obviously increases the cost. In addition, as some new registers are added, it implies the operating system must be modified to support the extensions. The two other choices, on the other hand, do not need any additional support from the operating system. This various pros and cons, explained thereafter, are discussed in [14].

Using the floating point data path and registers is interesting as it leaves the integer data path free to perform the control logic (address and branch computation). In addition, on 32 bit processors, floating point registers are larger (64 or 80 bits) than the integer registers (32 bits), allowing to pack more data elements in a single register. Moreover, many of the multimedia instructions are multi-cycle instructions, and thus fit best in the floating point data path. Putting them in the integer data path would very likely increase the critical path, thus lowering

the overall performance.

Nevertheless, Hewlet-Packard chose to implement its MAX-2 extension on the integer data path, for several reasons. Indeed, having the extension use the floating point data path and registers can create a performance issue if floating-point and multimedia extensions are to be used together; and designers found that a variety of multimedia applications, such as 3D, are floating point intensive. Moreover, they felt this would produce a more compact implementation, outweighing the various advantages. Indeed, many parts of the integer execution unit can be reused.

1.3.2 Supported data types

As it has already been pointed out, data elements used in multimedia applications are usually small, allowing more than one element to be packed in a single register. The register can then be considered as a vector, containing a few scalar elements. However, the available data types, as well as the vector size, vary greatly from one platform to another.

Obviously, bigger vectors allow packing more elements in a single vector, thus performing more operations at a time. However, having bigger registers and performing the operation on more elements at a time also increases the gate requirement and price. As a consequence, there are two kinds of multimedia extensions: those using 64 bit registers (MMX, MDMX, MAX-2, VIS) and those using 128 bit registers (AltiVec, SSE, SSE2).

Another difference is related to the type of data handled. All multimedia extensions can perform operations on integer data types. However, more advanced extensions also perform operations on packed floating point data types: this is the case of SSE and SSE2.

The size of the data elements is also an important consideration. The various data element sizes are 8, 16, or 32 bits for integer elements. Floating point data types may be 32 and 64 bits wide. Depending on the extension, more than one kind of data type may be supported, providing more flexibility and optimal performance and flexibility. Others support only one kind of data type, making the extension much simpler and easier to integrate in the chip.

1.3.3 Supported instructions

As for the supported instructions, they can be divided into four groups:

- arithmetic instructions;
- data conversion and reorganization instructions;
- memory access instructions;

- special instructions.

Obviously, all vendors support arithmetic instructions as part of their multimedia instruction sets. Indeed, new instructions are needed in order to take full advantage of the extension by performing operations on packed data types. Addition, subtraction, multiplication and division of subword operands are always supported. These operations usually support signed and unsigned overflowing arithmetic. For improved performance, some of the architectures also support saturated arithmetic, which allows larger results to be clamped to the largest available value. Some vendors also offer less common instructions: MAX-2 includes an arithmetic mean instruction; MDMX and MIPS V support vector/scalar multiplication⁷. Finally, some instruction sets allow selecting the subwords on which the operation is to be executed.

An important limitation related to the SIMD model used here is that the data needs to be packed in the register before it can be used. This can add an important overhead and degrade significantly the overall performance of the algorithm. This is especially true if the data needs to be converted to another data type, or if there are some complex dependencies between the elements. To minimize this degradation, all vendors support a variety of packing and unpacking operations. These allow moving 8, 16 or 32 bits of data to or from a multimedia register. Some vendors also include special instructions allowing to permute subword inside a register, or to merge subwords from different registers.

Obviously, all vendors support load and store operations, allowing filling the multimedia registers with the data and later putting the result in memory. Usually, the operation copies an entire register, although some vendors also include partial load or store instructions. Depending on the vendor, the load/store may have to be performed at an aligned memory location. SUN also provides a powerful feature: 64 bytes load and store. In addition, this transfer bypasses the cache, thus not displacing any useful data from the cache.

Finally, each vendor has added special instructions. Some of them are needed to support the new architecture. Some other are more specialized instructions, which the designers found would be useful and increase the performance. Most of the time, they are but of no use except for graphics applications. SUN, for example, adds an instruction to calculate the pixel distance, and another one to calculate a 3D array offset with a cache-friendly logic. These instructions can be very powerful, but have a quite narrow range of application.

1.4 Feature comparison

This section tries to present an overview of the features offered by the various architectures and multimedia extensions. Table 1.1 presents an overview of the

⁷Multiply the vector in one operand by one of the subword of the other operand.

features offered by some of the extensions. AMD 3DNow! is not included, as it provides about the same functionalities as MMX. Concerning the Intel platform, it is important to notice that the various extensions are cumulative: MMX was added to the Pentium II processor; Pentium III also included SSE instructions; and Pentium 3 adds support of SSE2 instructions.

Vendor	Motorola	DEC	SGI	Intel			Sun	HP	Intel
Processor	G4	Alpha	MIPS V	Pentium II, III and 4			SPARC v9	PA-RISC	Itanium
Extension	AltiVec	MVI	MDMX	MMX	SSE	SSE2	VIS	MAX-2	-
Vector Reg.	32	32	32	8	8		32	32	128
Reg. Size	128 bits	64 bits	64 bits	64 bits	128 bits		64 bits	64 bits	64 bits
Reg. Type	Dedicated	Integer	Float	Float	Dedicated		Float	Integer	Integer
Operands	3	3	3-4	2	2		3	3	3
<i>Supported packed data types</i>									
8-bit int	16	8	8	8	-	16	8	-	8
16-bit int	8	4	4	4	-	8	4	4	4
32-bit int	4	-	-	2	-	4	-	-	-
32-bit float	4	-	2	-	4	-	-	-	2
64-bit float	-	-	-	-	-	2	-	-	-

Table 1.1: Multimedia Instruction Sets Features

1.5 How to use it ?

Although multimedia extensions appeared more than 5 years ago, using them is still quite difficult. Indeed, a programmer who would like to use them to get the best performance for his applications has only a few choices. Obviously, he can spend some time to hand-optimize the assembly code, so that it makes use of the “new” compiler extensions. Although this solution is certainly the one giving the best results, it is also the most difficult one, and is certainly not portable.

Another option is to use in-line assembly or compiler intrinsics provided by the compiler vendor. This method still requires the programmer to have a deep knowledge of the architecture and instructions, but has the advantage of letting the compiler perform the register allocation. Some vendors (Intel...) also provide higher level abstractions, on the form of template classes. While providing good performance, this method still requires a good understanding and knowledge of the system by the programmer, and thus does not simplify the optimization either.

Depending on the application and actual offering, the programmer may have the option of using a third-party library which implements the required high level functionalities and is already optimized for the target processor (using multimedia extensions). Such libraries are usually provided by the processor vendor, for some specific application fields: for example, SUN provides a signal processing library that makes use of the VIS instructions of its SPARC v9.

The other option would be to let the compiler do the actual optimization. This would be easy for the programmer, as he would not need to do much. Un-

fortunately, this method has not yet been implemented in commercial compiler. In addition, it can be noted that even if some techniques can be used to detect parallelism, compilation may never be able to perform as fast as hand-coded assembly do, because finding algorithm transformation making the best use of all available instructions is certainly not an easy task: for example, it does not seem easy to automatically decide if saturated arithmetic should be used instead of overflowing arithmetic.

Chapter 2

Existing Compilation methods for multimedia instruction sets

A number of techniques have already been used in research compilers to perform automatic generation of multimedia SIMD instructions. These techniques rely widely on the research made on SIMD computers a few decades ago.

2.1 Vectorization

All the multimedia extensions added to general purpose processors follow the same architectural paradigm: they provide some sort of small scale SIMD execution support. As a consequence, the techniques used for SIMD processors can be used to generate optimized code using these extensions. One such technique, known as vectorization, can indeed give interesting results when implemented for MMX [15] or VIS [16, 17]. Even though they are clearly sub-optimal, the results are quite promising, with speed-ups ranging from 1.5 to 6.5.

The main motivation behind vectorization is that in computation intensive applications such as multimedia applications – where some processing is applied to large data sets containing small elements – loops are the most critical part of the code and should present a large amount of parallelism. Thus, one solution to optimize the whole application is to detect these loops that can be parallelized, and transform them into a vector operation, operating on infinite length vectors. This operation is then transformed in a loop using the SIMD operations.

It is in fact quite simple: first, some transformations are done on the loop to give it a standard form, which can be vectorized. Then dependency analysis is done to verify the validity of the vectorization. Finally, the vectorized code is optimized to minimize overhead. Classic vectorization is thus done by performing the following steps sequentially:

- *loop analysis*: determines the loops;

- *loop normalization*: adjusts the iteration count to start at 0 by a step of 1;
- *dependence analysis*: computes the data dependence graph of the loop;
- *scalar expansion*: transforms a scalar variable used inside the loop into a vector;
- *loop distribution*: distributes the loop over the strongly connected components in the data dependence graph;
- *vectorization*: transforms the loop body into vector instructions;
- *strip mining*: reduces variable length vectors to vectors of the available size;
- the assembly code can then be generated, registers allocated, and classical optimizations applied to improve performance even more.

Each of these steps are described in details in [15, 17]. There are some important points to detail, though. First, one also needs to handle alignment and boundary conditions properly. Indeed, multimedia instructions may not work – or work slower – with unaligned memory locations. It may thus be needed to perform special cases before and after each vectorized loop. This can be a problem if the operands are not statically known. These issues are handled in great details in [16].

Another problem is that of format conversion. Indeed, multimedia instruction sets usually support more than one packed data type. Thus, some operations may have to be generated to perform the conversion between one type and another.

Finally, one must handle conditionals properly. One solution is to flatten the conditional operations before the vectorization. All conditional may be evaluated before the actual calculation is done, and the result stored in a temporary. This approach is used by G. Cheong and M. Lam in [16]. In [17], A. Krall and S. Lelait follow a similar approach, where the vectorizer introduces a mask in the iteration space of each vector statement. A vector comparison is generated for each atomic condition, which are then combined together. This can then be used as a mask for a partial store. Thus, this method is optimal only if partial store is supported; else, some more work may be needed in order to simulate the partial store, which may degrade the performance more.

Some variations exist on this algorithm. In [15], N. Sreraman and R. Govindarajan propose to perform the strip mining step before the vectorization. This provides the same results, but makes the vectorization work directly with the actual vectors. There is no need to infinite vectors anymore.

This approach is in fact a first step toward that introduced by A. Krall and S. Lelait in [17]: he proposes to perform *vectorization by unrolling*, which is way simpler to implement. The main idea is that instead of performing a full vectorization, one can simply unroll the loop a few times (as many time as the number

of operations the instruction set can perform in parallel on the smallest precision variable used in the loop body). Then, instruction scheduling is performed to group together instances of the same loop instruction, according to the data dependence graph. Vector instructions can then be generated.

Although it is a classical technique, actually one of the first used for parallel computing, vectorization had never been used for general purpose processors. It is now being used to generate optimized code for multimedia instruction sets, but only in research projects. Thus, not all optimization are performed, which make it not so efficient. Only a “limited” vectorization is implemented, meaning only the inner loop is vectorized. Some other loop transformations (loop interchange, loop splitting) may be added to find more parallelism. Also, complex loops and non-looping code may not be dimmed vectorizable, and hence not optimized, whereas it can present opportunities for multimedia SIMD instruction.

2.2 Superword Level Parallelism

Although it is certainly well known and gives quite good results with a number of multimedia applications, vectorization also resents a few shortcomings. Indeed, many important multimedia applications are difficult to vectorize. In addition, complicated loop transformation techniques, such as loop fission and scalar expansion, are required to parallelize loops that are only partially vectorizable. In [18], S. Larsen presents an alternative way of extracting parallelism. He denotes “Superword Level Parallelism (SLP)” this form of parallelism, more suited to short SIMD operations than vector parallelism. It is defined by the author as follows:

Superword level parallelism is defined as short SIMD parallelism in which the source and result operands of a SIMD operation are packed in a storage location.

An advantage of SLP over vectorization is that it does not require large amount of parallelism to be profitable.

Somehow, SLP can be seen as a restricted form of ILP¹. Indeed, executing the same instruction on all subwords of a packed register is about the same as executing a few instructions at the same time, each performing the operation on one of the subwords. Just as ILP, SLP can be used to extract parallelism from basic blocks. It can be noted also that this technique can be used to make an elegant implementation of a vectorizing compiler: by unrolling the loop and applying the SLP extraction algorithm to the loop body, one can vectorize the code, without any complex loop transformation.

¹Instruction Level Parallelism.

Detection of SLP is quite simple: one just needs to find the independent, isomorphic² statements within basic blocks, and group them together. Such statements can be executed in parallel, thus source operands are packed in registers, and instructions are replaced by their SIMD equivalents. Results also need to be unpacked after the calculations.

One problem however is that there may be several different packing possibilities within the same basic block. S. Larsen proposes in [18] an optimal algorithm to extract SLP. Unfortunately, his approach is far too complex to be computable, which is why he also gives an heuristic.

It first tries to pack statements 2 by 2, starting with isomorphic independent statements that refer to adjacent memory locations. Thus, it builds a set of tuples, each composed of two statements. In the whole set, a single statement may not appear more than twice: once as the first element of a tuple, and once as the second element of the tuple. This set is then extended by following def-use and use-def chains of existing entries. Finally, tuples containing the same expression are combined, allowing scheduling them as SIMD operation. Scheduling must also take care not to produce a dependence violation. This is done by building a dependence graph for the groups: as long as there are no cycles, all groups can be scheduled. If that is not the case, at least one group needs to be eliminated in order to ensure correctness.

To make the algorithm more efficient, a few steps are needed. First of all, one can notice the algorithm can be applied at almost any level. For better results, statements should be flattened into three-address form, to create more statements with common operations. Inner loops should also be unrolled, to expose loop parallelism. Also, most common optimizations should be applied before SLP extraction, to ensure SLP does not parallelize code that would otherwise be eliminated or greatly simplified. Scalar renaming is especially useful, as it removes some of the dependencies that would otherwise inhibit parallelism.

2.3 Pattern matching

An interesting approach to parallelization of code is that taken by M. Boekhold, I. Karkowski and H. Corporaal in [19]. Instead of having a compiler with multiple passes, they propose to have a single transformation engine, which can replace code matching a specific pattern by another code fragment, provided some user-provided conditions are respected. This way, the parallelization can be extracted quite simply, by providing the correct transformation specifications.

Although they do not give any practical example of such transformation, R. Manniesing, I. Karkowski and H. Corporaal give more details on a SIMD transformation specification in [20]. There, it appears the transformation matches only very specific cases: a very specific operation pattern must appear in the

²Statement that contain the same operations in the same order.

program. To make it more efficient, it could certainly be combined with other transformations (specified the same way), which would disclose more “parallelizable” patterns. However, it already provides good performance, finding 85% of the vectorizable loops according to the authors.

Obviously, a “problem” with this approach may seem to be that one must create transformation for each kind of operation. This is definitely an issue, yet it also allows developing the optimizer in an incremental fashion. Many transformations would also certainly look alike, and thus the effort for writing them will be far less than that for writing the first one.

Finally, an interesting prospect of this approach is that it can make far better use of the available instructions. Where vectorization or superlevel word parallelism could only extract parallelism, this method could also be used to generate more “classical” optimizations or to generate code making use of specific multimedia instructions: multiply/add, saturated arithmetic. . . This certainly would not be an easy task, but at least this method offers an opportunity.

2.4 Code selection

In [21], Rainer Leupers presents an alternative method, which can be easily added to current production compilers. Instead of using some source-to-source transformations (or equivalent methods) to generate compiler intrinsics recognized by a back-end compiler, he shows a method allowing to integrate SIMD code generation into the compiler’s code generation phase.

In most compilers, code selection is done by mapping the available instructions on the data flow tree (DFT). Thus, techniques of tree pattern matching with dynamic programming are often used, as they can compute an optimal covering of each DFT in linear time.

This approach is of little help for the generation of SIMD instructions, as this requires the simultaneous covering of multiple DFTs. Indeed, it is often needed to pack together operations located in different DFTs. Thus, code selection usually needs to be performed on complete data-flow graphs (DFGs).

To achieve this, Rainer Leupers proposes to partition a given DFG into multiple DFTs. Then, each DFT is analysed to generate alternative solutions: multiple optimal solutions are detected, including solutions making use of SIMD instructions; a global, optimal solution can then be computed for the whole DFG.

In order to do this, the algorithm considers the various ways to use a register: a 64 bits multimedia register could thus be considered as two 32 bit registers, as well as four 16 bits registers, or height 8 bits registers. At this time, the actual register is not considered as a single entity: the algorithm considers it virtually as if it were a few distinct registers.

SIMD instructions are also described accordingly: an SIMD instruction allowing to simultaneously add two 32 bits values would be described as being able

to add the lower half of two multimedia registers to put it in the lower half of a third one, and to perform the same thing on the upper half of the registers. Both of these patterns can be used independently in the various alternatives to cover the DFG nodes (the DFTs). This way, tree pattern matching techniques can be used to cover each of the DFTs independently, while already generating, among other alternatives, SIMD “sub-instructions”.

The second part of the algorithm consists in generating the global covering of the DFG. This is done by selecting the actual alternative for each of the DFG nodes, in order to maximize the use of SIMD instructions across the entire DFG. This is done by formulating the problem as an Integer Linear Program.

In this problem, each alternative of each node in the DFG is represented by a Boolean variable. On top of this, constraints are added to ensure that:

- the solution to be valid: each node is covered by exactly one of its alternative coverings;
- data types expected on both end of a dependence arc are the same: if the rule selected for a node specifies that one of the argument is in the lower half of a 64 bit multimedia register, then the rule for the corresponding son in the DFG must generate something in the lower half of a 64 bits register;
- CSE³ are used consistently: if a rule is used to cover the use of a CSE, then all uses of that CSE must be covered by this rule;
- SIMD sub-instructions are packed correctly into SIMD instructions. The author thus introduces the notion of “SIMD pair”⁴. A pair of DFG nodes is a SIMD pair if there is no scheduling between the nodes, they have the same operator, the first node can be located in the upper part of a register, and the second one in the lower part. Moreover, if they represent load and store operations, then they refer to adjacent data in memory. The constraint guarantees that each selected SIMD instruction actually covers a pair of DFG nodes, and that any DFG is covered by at most one SIMD instruction. To do this, new Boolean variables are introduced for each SIMD pair, which denote that the nodes are packed together into a single SIMD instruction;
- there is no scheduling deadlock: if a SIMD pair \mathcal{P} is covered by a SIMD instruction, then SIMD pairs where one of the nodes must be scheduled before one of the nodes of \mathcal{P} and the other after one of the nodes of \mathcal{P} should not be packed together.

³Common Sub-Expressions. In this systems, CSE are strictly assigned a register, and register read/write nodes are added into the DFG to replace edges

⁴In [21], the author considers a target processor which can only achieve 2-way sub-word parallelism. The notion can be easily generalized for greater order parallelism.

The above constraints ensure that the solution is correct. To optimize the solution, the author tries to maximize the number of selected SIMD instructions across the entire DFG. Thus the objective function is simply the number of selected SIMD sub-instructions. An ILP solver is then used to discover the optimal solution.

The major advantage of this technique is that it could be very easily integrated into current compilers, thus making it easy for many applications to transparently use SIMD extensions. As it does not require major changes to the compiler's architecture, it could be smoothly integrated into current production compilers: his algorithm can be implemented as a step-in replacement for the existing instruction selection algorithm.

Moreover, all the processing needed for generating the SIMD instructions is mostly processor independent. The techniques used to describe a processor's instruction set can be reused with small changes for the most part, thus allowing the compiler to be cross-platform.

The drawback however is that the calculations can take quite some time if the code is complex, namely if the DFG is large. To overcome this, and balance optimization and compilation time, the author proposes to let the user specify a threshold value for the size of DFGs passed to the code selection engine.

Chapter 3

Machine abstraction

3.1 High level SIMD execution model

As we have seen, the multimedia SIMD instructions use partitioned registers, allowing a single register to be considered as a vector. This way, instructions operating on each element of the vector can be implemented at a small cost. Each of the microprocessor vendors has provided his own extension, and most of today's architectures include such operations. Each of the vendors tried to find out the minimal set of operations and data types that would provide the best performance. As a consequence, all multimedia extensions provide some common functionality, even if all of them also provide more specific features. However, many operations and data types which could be useful for certain applications are not provided. This stems from the fact that the current SIMD extensions were designed for specific multimedia applications and algorithms.

In [1], R. Fisher presents a bigger set of SIMD instructions, which would not be biased in favor of multimedia applications. Designed with portability in mind, he proposes a set of higher level SIMD constructs, supporting various data type widths, and which can be mapped directly to existing hardware SIMD instructions or emulated in software. Thus, [1] also includes efficient implementation of those instructions using regular or SIMD instructions. As a consequence, it is possible to offer a consistent, complete and portable interface to SIMD extensions. If a program is written using this interface, the compiler can generate the actual code for the target architecture easily: it simply needs to know how to implement each of these functionalities on the target architecture. In addition, R. Fisher presents a few optimizations that could be used to improve the speed of the missing operations.

Thus, R. Fisher defines a few classes of operations, and gives guidelines on how to map them to existing processor instructions:

- *polymorphic operation*: operations which can be implemented independently of the field partitioning (for example, logical operations). These

operations can be implemented with the associated, non-SIMD, instruction;

- *partitioned operation*: operations which operate on each element of the vector independently (for example, arithmetic operations). Operand fields must be manipulated without interfering with the other fields. They must be implemented in software in most cases, as current hardware implementations support only a few partitioning schemes;
- *communication operation*: operations which transmit data across elements in an arbitrary pattern. There is almost no hardware support for these operations, so they must be implemented using unpartitioned shifts and logical operations;
- *type conversion operation*: operations which convert data from one type into another. This can imply some problems as the number of elements may not be the same from one type to the other. Many type conversion operations are supported in hardware, and other can be implemented easily using communication and partitioned operation;
- *reduction operation*: operations which recursively combine fields values into a single value. There is no such concept with traditional SIMD architectures, but can be implemented easily;
- *masking operation*: operations which allow some elements to be excluded from the computations. This may be supported in hardware, or can be achieved using some arithmetic techniques;
- *control flow operation*: operations which perform a branch. The element must be moved into a generic register in order to perform a regular branch operation.

3.2 High level description of the available parallel instructions

In order to make the code target independent, another option may be to create some machine independent type extensions. Operations on these types would then be implemented and optimized for each of the target architecture. In [22], P. Cockshott presents a way of specifying the architecture that allows him to automatically generate this target specific implementation. Using this technique, a code generator could be automatically re-targeted to a different instruction set, based on the formal description of the instruction set.

In order to do so, he creates a language, ILCG, which does the mediation between the instruction set for a specific CPU and the high level language program.

It is used to generate automatically a code generator, which can be used by the other parts of the compiler. Based on the information in the ILCG source, some pattern matching techniques can be used to find out the instruction to generate.

Unfortunately, the author concentrates most of his efforts on the ILCG language definition, and only gives an overview of the various operations he performs to get the code generator. The whole application is available as part of the free *Vector Pascal* compiler, implemented in Java.

Chapter 4

A simple vectorization method

4.1 Overview

In our system, the vectorization is done by sequentially applying a number of phases, each performing some specific code transformation or analysis. The first few phases are usual phases, designed to make the core vectorization process simpler and more efficient.

1. Atomize
2. Compute “optimal” unroll factor, and unroll correspondingly
3. Detect and eliminate reductions
4. Transform into single assignment form
5. Reorder and group statements: analyze code to detect isomorph statements, and regroup them if possible
6. Vectorization the detected groups, and analyze SIMD vector dependences
7. Generate SIMD code

Each of these steps is described in detail below.

4.2 Atomizer

Most multimedia extensions include only “simple” operations: additions, multiplications, and so on. Therefore, there is no point in keeping the expressions in their original, compact form. In order to simplify the remaining of the process, we thus atomize the expressions, so that all statements are simple, two or three operands expressions. This would be needed eventually anyway, when generating the SIMD instructions, and simplifies greatly the following phases.

This has an additional side effect: by splitting possibly large expressions into smaller, simpler expressions, we are more likely to have operations that can be executed concurrently, and are thus potential candidates for SIMD vectorization. Indeed, if we have a few complex expressions, they may not perform the same basic operations, and thus would not be parallelized. However, computing both expressions is likely to rely on the same kind of simple operations: additions, multiplications, . . . Thus, part of the calculations may be done using SIMD extensions, instead of being done sequentially.

It is important to notice, however, that this process somehow loses some information: it loses the information that a set of calculations is in fact used only to get a specific result, and that the intermediary variables are needed only for this calculations. Yet, this is not really an issue in our case, as our algorithms do not rely on this, and the information is rebuilt later in a more generic fashion anyway.

Nevertheless, there is a piece of information we need to keep. The issue regards all references. A reference is a variable, and for array variables also includes the indices for each of the array dimensions. It is up to the atomizer to atomize the indices if they are too complex. It is important here not to atomize too much. Indeed, if the reference to array A in:

$$y = f(A[x + 1])$$

were atomized, resulting in:

$$t = x + 1; y = f(A[t])$$

then it would be almost impossible to detect adjacent memory references¹. It is therefore important not to atomize subscript expression of the form:

$$\textit{scalar_reference} + \textit{constant_offset}$$

4.3 Optimized unrolling

Many multimedia applications perform repetitive operations on small quantities. An obvious example is the computation of the sum of two vectors. However, even if the process itself is obviously parallel, the code does not make this parallelism obvious. To increase the “visible” parallelism level, the code can be unrolled.

This transformation has many advantages. First, it can always be done. It is always possible to unroll a loop. Care must be taken to ensure that all variables have the same value after the loop, and that all iterations are computed. This may generate some overhead, especially if the range is not a multiple of the unroll

¹References which refer to adjacent locations in memory.

factor, or if it not known at compile time. This overhead is however not significant if a sufficient number of loop iterations is executed.

Moreover, this transformation makes parallelism explicit. If this transformation is done appropriately, all the available parallelism can be present in the loop body. This lets us greatly simplify the algorithm: after this is done, it is not necessary to analyze loops to discover parallelism, and we can assume all available parallelism is present in the instructions sequences. As a side effect, this allows the algorithm to work not only with loops, but also with any sequences of statement.

As a consequence, a good unrolling strategy is needed in order to leverage the maximum performance. In our implementation, we provide two different strategies, and let the user choose the one he prefers.

A simple algorithm

The first one is a very simple algorithm, but quite efficient nonetheless. It simply looks at the variables used inside loop bodies, and more precisely at their width. This way it finds out the maximum and minimum width required for variables. The unroll factor can then be computed simply by the simple formula:

$$\text{unroll_factor} = \frac{\text{vector_size}}{\text{variable_width}}$$

Still the choice must be made between using the minimum or maximum of the variables' width. This choice is not as trivial as it seems. Obviously, using the minimum will induce a bigger unrolling factor, thus giving access to more parallelism. However, this may add some overhead as additional instructions will be needed to convert vectors from one packing mode to another. We believe using the minimum variable width will give more consistent results, and that the different may not be that important, for a few reasons.

First, this makes the vectorization less likely to break the algorithm structure. By keeping the unrolling factor low, we can hope that all operations will get vectorized the same way. Moreover, the performance gained by unrolling more is not that important, even if there is definitely some improvements. In addition, this code, supposedly more optimized, may eventually be crippled by the number of available registers, especially on Intel processors: in this case, additional instructions would be added to load and save register values to and from memory. For all those reasons, the algorithm uses the minimum variable width by default. One may try using maximum width, as it may give better performance.

A more precise algorithm

Another algorithm can be used to compute the unroll factor. The advantages of this algorithm are that it does not need to know the vector size, and that it

takes the available instructions into account. However, it is most certainly slower than the previous one. As opposed to the previous algorithm, this algorithm is less likely to unroll too much.

This algorithm is in fact quite simple. The idea is to analyze the statements inside the loop bodies, to see what instructions could potentially be used by the vectorizer. Obviously, only simple checks are done here: we simply look that SIMD operations which match the calculation needed, and check that they can accommodate the required size for the arguments. As a result, it is easy to see the maximum level of parallelism that can be obtained for each statement. The unroll factor stems easily: the loop should be unrolled so many times as to get the maximum level of parallelism.

4.4 Reduction detection and elimination

If not handled properly, reductions can cripple the vectorization. Indeed, a reduction introduces dependences between each of the implied statements, therefore preventing any attempt of vectorization. Hopefully, reductions can in most situations be parallelized, by performing some simple transformations.

The first thing to do is to detect the reductions. This is done using a unified semantic approach, described in great details in [23]. The framework allows the detection of *generalized reductions*, which include a wide range of programming idioms: loop invariant, induction variables, and reduction operations. However, the implementation we use only detects regular reductions: sum, product, and, or, min and max operators are supported, as well as dead reductions.

Once all detections have been detected using this algorithm, we have, for each statement, the list of the reductions it performs. We can then traverse the code to get access to the loop enclosing all those reductions. Each loop is then transformed to minimize the dependencies and allow efficient vectorization of its body. It is important to notice that this transformation is done after the loops have been unrolled: therefore, the loop body of a reduction should contain more than one instance of the reduction, which would be easily parallelized if not for the dependencies between each of the instructions.

To do so, we first create a new vector variable for each reduction variable². The number of element in the vector is the number of reduction related to this vector³. In the example of figure 4.1, the variable `SUM` is used in four reductions: a new four-element vector is thus created, `SUMV`. Obviously, we consider reductions this way only if the variable is used in more than one reduction, and if those reductions use the same operator.

²Here we consider more a reference than a variable: it could as well be a specific element of an array, as long as the indices do not change in the context of the concerned loop.

³Note that a reduction, as detected by the previous algorithm, is bound to a single statement. If there are two identical statements, two reductions will be detected: one per statement.

<pre> for I=0 to N step 4 SUM = SUM + A[I] SUM = SUM + A[I+1] SUM = SUM + A[I+2] SUM = SUM + A[I+3] end for </pre>	<pre> ## Prelude SUMV[0] = 0 SUMV[1] = 0 SUMV[2] = 0 SUMV[3] = 0 ## Modified loop body for I=0 to N step 4 SUMV[0] = SUMV[0] + A[I] SUMV[1] = SUMV[1] + A[I+1] SUMV[2] = SUMV[2] + A[I+2] SUMV[3] = SUMV[3] + A[I+3] end for ## Compacting phase SUM = SUM + SUMV[0] + SUMV[1] + SUMV[2] + SUMV[3] </pre>
(a)	(b)

Figure 4.1: Sample sum reduction, before (a) and after (b) reduction elimination

Each element of the vector will be used to perform a share of the computation done in the loop. Some code will be added before and after the loop to initialize the vector and compute the final result, as expected. Figure 4.1 presents a sample code, as well as the associated generated code.

Prelude: the vector is initialized, with all elements set to the neutral element for the reduction operator.

Loop body: all references to the reduction variables are replaced by a reference to an element of the associated vector. In the first reduction, the variable is replaced by a reference to the first element of the vector, in the second one, by a reference to the second element, and so on.

Compacting code: this code is added after to loop to perform the calculation of the actual result of the reduction. To do this, it simply uses the reduction operator to reduce all elements of the vector—which are partial reductions—and the original reduction variable—to take the original value into account.

4.5 Single assignment form transformation

In order to minimize dependencies, and thus increase the vectorization potential of the code, yet another transformation is performed before the actual

vectorization. The goal of this transformation is to remove some dependencies that are due to a variable being reused. Those situation can be solved by simply renaming the variable at some point.

A typical case would be a temporary variable, which is used more than once. Something will be written to the variable, and it will be used in some calculations. Later on, another value is written to it, and it is used in another calculation. This rules out vectorization: the fact that they use the same variable precludes simultaneous execution.

However, it is easy to remove such dependencies by simply renaming the variable at some point. It is used in some calculations; instead of using it in later calculations, another variable is created and used in place of the original one. This way, as many variables are used as there are assignments, and the variable does not prevent vectorization anymore.

This phase is especially important, since such “fake” dependencies are created during the unrolling phase. The algorithm we developed performs this transformation in two steps. First, as the dependency graph implemented in Pips does not include a list of predecessors, we traverse the whole dependency graph to find out the number of predecessor for each reference. This will be used in the second stage of the algorithm to prevent the transformation in situations where it may break the code.

When this is done, the dependency graph is traversed one again, this time to do the actual transformation. For each node, we look at each reference it writes and which is read at some other place. If for all such arcs, the target reference⁴ has a predecessor count of one, we create a new variable and replace the variable in the original reference as well as all the dependencies with this new variable. If a single reference has a predecessor count greater than one, then we forfeit the transformation for this reference.

The test we use is very conservative, but allows the transformation to be performed simply, as updates do not need to be cascaded. Indeed, if the predecessor count for a reference is greater than one, then the references used in the various predecessors statement are connascent⁵, although this information is not shown directly in the dependence graph. This implies that if we modify (rename) the reference in one of the predecessors, we must modify in all the predecessors for the code transformation to be correct.

For simplicity, we decide to do nothing in such situation, which allows making things simple: there is no need to go and modify all other definition sites. It is true however that some dependencies may be removed by doing so, with a more complex algorithm and at the expense of some computation time: indeed, we would need to build a full graph of predecessors, and most importantly, to update

⁴the reference at the other end of the arc, which is read

⁵from the Latin: “having been born together”, and, by extension, “having inter-winded destinies in life”

all the dependencies of the other definition sites, and so on. This could easily lead to cycles in the processing, so lots of care would be needed. Moreover, our greatest need is to remove the fake dependencies introduced by the unrolling of atomized code. Our algorithm, as is, removes those dependencies, and certainly even some dependencies from the user's code, so we can consider this sufficient.

4.6 Statement reordering and grouping

The previous phases are used to increase the potential for parallelization, yet do not perform any kind of vectorization. The vectorization itself is performed in three steps:

- Statement reordering and grouping.
- Group vectorization & SIMD vector dependence analysis
- Optimized load computation & SIMD code generation

The whole process is based on the notion of *vectorization group*. A vectorization group is a group of statements that may be vectorized as a single entity. This means the statements are independent and isomorph, i.e. all statements in the group can be executed at the same time, and the whole group can potentially be replaced by a set of SIMD instructions. Note that a group may contain too many instructions to be replaced by a single SIMD operation. Anyway, a first step is to build the groups. Later on, groups will be mapped to SIMD operations, and finally SIMD instructions (including necessary load and save instructions) will be generated.

The algorithm used to detect a group is fairly simple. It consists in a greedy algorithm, which operates on sequence of statements, as defined in PIPS internal representation. The algorithm does not detect, nor try to detect, groups that are in different statement sequences. Among other things, this means it does not work across loops boundaries, although it may try and detect groups from statements originally executed before and after a loop, as long as these statements are in the same sequence (and if this does not break dependencies); however, statements from different loop bodies will never be grouped together.

First, each statement is matched against a set of expression patterns. This allows a list of patterns which apply to this statement. Each of those patterns describes an operation, and is associated with a set of SIMD opcode. The details are described in further details later on. The patterns, the associated operation classes and the opcodes are platform-specific, and defined in a configuration file.

Groups are then constructed incrementally by looking at each of the statement in the sequence, in order. The first statement is added to the group. The set of matches of the group is then set to the set of matches for this first statement. All the statements following this statement are then examined in turn.

If the set of matches for the group and for the later statement do intersect, then the statement may be added to the group. However, this is valid only if the statement is independent with all the statements since⁶ the first statement of the group. If that is indeed the case, then the statement is added to the group: it is moved at the end of the group, and the set of matches for the group becomes the intersection of the matches of all statements in the group.

Once all statements following the group have been considered, all the statements that have been said to be part of the group have been moved right after the first statement of the group. We then consider the group as complete, and start a new group from the first statement following the group. By construction, we are guaranteed that we did not yet find a group where this statement belongs. The search goes on till all statements in the sequence have been put in a group.

Once a group is found, two situations can arise: or the group is composed of a single statement, or there are more than one. If there is only one statement, then this statement is added to the generated sequence of statements. In the other situation, the group is vectorized (if possible) and the resulting statements are added to the generated sequence of statements. This process is explained in the next sections. Finally, the original sequence is replaced with the new sequence of statements.

Matching algorithm

Our algorithm has been designed to match complete statements, one at a time. Therefore, it boils down to simply finding if a tree, the statement's expression tree, maps to one of the patterns' expression tree. This can in fact be performed quite efficiently, with a single traversal of the statement's expression tree.

To do so, each pattern is associated with a list of symbols, each corresponding to a specific kind of node in the PIPS internal representation. Symbols can be operators, or one of two special symbols, for constants and references respectively. In these conditions, a pattern represents the list of symbols that would be generated by a pre-order traversal of the expression tree the pattern matches.

The algorithm is then quite simple: all the patterns are put in a matching tree, where each node is tagged with the set of matched patterns, and each edge with a symbol. To match a statement, the only thing needed is to perform a pre-order traversal of the tree. At each step, the symbol generated by the traversal is matched against the tag of the edges of the current node. If there is no matching edge, then there is no match. Else, the current node becomes the one pointed to by the matching edge, and the traversal goes on. Finally, when the traversal is over, the current node lists the set of matching patterns.

The algorithm allows getting multiple matches for a single statement. The interest is to detect that a statement like $y = x$ can also be seen as $y = x + 0$ or

⁶With respect to the sequence order

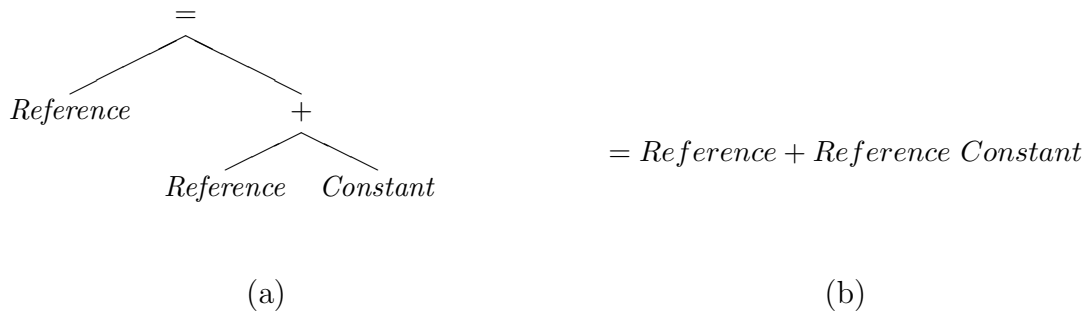


Figure 4.2: Pattern definition, as a tree (a) and as a sequence of symbols (b)

$y = x \times 1$. To handle this, each pattern is associated with a list of symbols, as we have seen, but also a list of arguments. This list is used to build a list of *effective arguments*, which is the list of arguments to be used if using the associated match, to get the same effect.

For example, the expression $y = x$ would be matched by the *copy* pattern, with arguments (y, x) . However, it will also be matched by the *add* pattern, with arguments $(y, x, 0)$, as $y = x + 0$, and by the *multiply* pattern with arguments $(y, x, 1)$, as $y = x \times 1$. The arguments list of the pattern is used to translate between the list of references and constants (i.e., arguments) used in the original statements, and the list of arguments that can be used by the pattern. Currently, this transformation is quite simple, and does not allow the reordering of parameters. It simply allows inserting constant parameters anywhere in the list of actual parameters.

To conclude, this algorithm is used to efficiently match the actual statement against a set of patterns, to identify a set of operations that can be used to implement it, and for each operation the arguments that should be used. To make it even more general, it could be modified to include guard expressions, which would be conditions that must be verified by the parameters for the match to be effective. This would for example allow to detect that $y = 2 \times x$ can be seen both as a product and an addition, or that $y = x + x$ can be seen as a multiplication by two. Support could also be easily added for tests, which could thus be used to detect *max* operations.

4.7 Group vectorization and SIMD vector dependence analysis

Once a group has been fully identified, the group vectorization is used to generate an efficient implementation using SIMD instructions. The algorithm takes as input the list of the statements of the group, as well as the list of the group's matches. It builds a list of statement information that can be used to

generate the actual statements.

First, the algorithm chooses one operations from the list of operations matching all statements in the group. As it is not likely there are more than one matching operation, we simply use the first operation in the list. The statements of the groups are then mapped on the available opcodes. Indeed, the groups built by the previous phase can contain arbitrary number of statements. This mapping is done using a simple method. It simply selects the opcode which:

- performs the required operation;
- has arguments wide-enough to accommodate the arguments;
- has the greatest parallelism level, i.e. can perform the operation on the greatest number of data elements.

In addition, we restrict the search to operations which do not have parallelism level greater than the number of scalar operations we need to do. This choice was made for two reasons. First of all, using operations that operate on more data elements than is actually required is very likely to generate additional overhead for store operations, as care must be taken to ensure the extra data does not overwrite useful data.

The second argument is simplicity. It may be that in some situations, performing more operations than required allows to reach better performance, as calculations are performed with fewer instructions, and there is not much overhead in this particular situation. However, the logic needed to detect those situations, and generate the optimized code to handle loading, is quite complicated, and will most likely not be useful in most situations.

For each actual SIMD operation that is scheduled this way, all arguments are put in an argument matrix. Each cell contains a data element, i.e. a reference, as we assume the code has already been atomized. Each column corresponds to a statement from the original code. Each line corresponds to an argument. Therefore, each line will eventually be mapped to a vector, or SIMD register.

SIMD vector dependence analysis

While the information is built up concerning the statements to be generated, another analysis is performed, the vector dependence analysis. This analysis is used to track the usage of the SIMD vectors and their contents. Later phases can thus use this information to find out if the content of a vector is already loaded in a register, or if it can be computed efficiently from other registers, using some specific instructions.

To do so, for each statement to be generated and for each data element used, we build the list of all *places* where it can be found. The places need to be precise enough to tell both the vector and the position in the vector. For convenience,

we do not store it like that, but refer to it as a position in the arguments matrix of another SIMD statement.

This information is built in a single traversal of the SIMD statements. To achieve this, we maintain a list of all the argument expressions encountered up to now, and of all the places where those expressions have been encountered. Notice that this algorithm only works because we are working on a single sequence of statements, which we are traversing in order.

When looking at an argument, two situations can arise. If the argument is read, then we know it is dependent on all the places where the expression can be found, as stored in the list. We also need to mark that the expression can also be found at this place. On the other hand, if it is written, then the list of the places where this argument can be found is reset.

We already explained that the statements must be traversed in order. Care must also be taken when traversing the argument matrix. Indeed, all “read” arguments must be processed before the “write” arguments, in order not to add non-existent dependency, making one argument of an instruction dependent on the result of the instruction.

Care must also be taken when non-SIMD statements are present in the sequence: indeed, when such a statement is encountered, it must be marked that all expressions which it has effect on are not available anywhere, to ensure the latest, correct value, is used (and loaded).

Note on reductions

Thanks to the reduction elimination phase, reductions presented to the core vectorization algorithm do not appear as such, and can be vectorized properly by the current algorithm. One glitch however is that the algorithm will generate load and save instructions inside the loop body, whereas only one load instruction before and one save instruction after the loop are enough.

Due to the way the vectorization algorithm works, it does never have access to both the loop body and the outer context. Therefore it is not possible to manage this directly during the vectorization, even if the information that some instructions correspond to a reduction is transmitted from previous phases. Using such information, we could however generate neither load nor save statements, and have a later step generate the load and save instructions needed for the reductions, outside each reduction loop.

Another solution would be to add a new generic phase to cleanup such move such load and save instructions outside of loops. As a side effect, this would also move constant load instructions outside of loops. Indeed, this phase would certainly be more generic, and would hopefully lead to better performance, but is certainly not trivial, and would thus require quite some work.

$\begin{pmatrix} A0 \\ A1 \\ A2 \\ A3 \end{pmatrix} = \begin{pmatrix} B0 \\ B1 \\ B2 \\ B3 \end{pmatrix} + \begin{pmatrix} C0 \\ C1 \\ C2 \\ C3 \end{pmatrix}$	<pre> ## Load statements VEC_B = VEC_LOAD(B0, B1, B2, B3) VEC_C = VEC_LOAD(C0, C1, C2, C3) ## Computation VEC_A = VEC_ADD(VEC_B, VEC_C) ## Save statement VEC_SAVE(VEC_A, A0, A1, A2, A3) </pre>
(a)	(b)

Figure 4.3: Sample vector operation (a) and its implementation (b)

4.8 SIMD code generation

Once the groups have been vectorized, it is time to generate the actual code. As we are working on a source-to-source optimizer, we do not generate actual assembly code, but rely on compiler intrinsics⁷. This has an added advantage: it lets us have as many registers as needed, which the back end compiler will eventually assign to the actual, available registers for the platform.

The code generation step is quite simple. It uses the information generated from the previous phase, which is a sequence of information on statements to generate. Two situations may arise: it may have to generate a SIMD statement or a “regular”, non-SIMD, statement. The situation is trivial for non-SIMD statements: the statement is simply added to the resulting code.

For SIMD statements, the situation is slightly more complicated. For each SIMD statement, a set of statements are generated, as shown on figure 4.3:

- one or more *load* statements, which load some expressions (as in the argument matrix) in SIMD vectors;
- a single *computation* statement, which performs the actual operation on the SIMD vectors;
- a *save* statement, which saves the result from the SIMD vector to the actual scalar variables.

Here we can see the main drawback of this programming model. As the multimedia instruction set is reduced, and as the instructions operate on distinct registers⁸, some overhead is added to move data in and out of the multimedia

⁷In fact, we do not rely directly on compiler intrinsics, but on a set of C macros which provide a slightly more convenient interface

⁸Logically at least: even if the same physical registers are used, the register cannot usually be used seamlessly for regular and multimedia, SIMD, instructions

registers. To maximize performance, some analysis is thus performed to ensure a minimal overhead. This can be achieved by using two main techniques: optimized load generation and unused save elimination.

Optimized load computation

To increase the performance, an important factor is the minimization of the overhead needed to load the data into the SIMD variables. A few situations can indeed be implemented more efficiently.

First, the best situation is when the data is already in a vector register, which can happen if the data has been loaded but not modified, or when the required data is the result of some vector calculation. In this situation, no overhead is needed to load the data, as we can directly use the already loaded variable. This is possible as we are working on a source-to-source compiler. Thus we do not take the number of registers into account. It is up to the back-end compiler to handle the actual number of registers.

Other situations are interesting as well. Indeed, some multimedia instruction sets provide additional instructions which support some conversion or reordering operations. Those operations can be used to build a vector when the data is present in another vector, but with a different order or packing.

In our system, the vector dependence tree is used to brutally search for an optimal load implementation. It currently detects aliased vectors (multiple vectors with the exact same content and packing style) and shuffled vectors (vectors with the same elements, but at different positions).

If all no solution is found to avoid the loading of data, a load operation is generated. We try however to generate an optimized load instructions. Thus, three situations can arise. If all expressions to load are constants, then we compute the value of the constant, and generate an instruction to load this value from memory.

We also check to see if we are not by any chance dealing with adjacent memory locations. This frequently happen after loop unrolling, as we access array elements with consecutive indices. Detecting this is fairly easy, as the indices are normalized thanks to the atomizer: an index is always of the form *reference + offset*. A set of references are detected as adjacent memory locations if they all access the same array variable, the reference used in the index is the same, and the index offsets are consecutive. The references must also be ordered correctly in the SIMD vector.

If all those conditions are met, we can generate a single load instruction, to load the required amount of memory from the address corresponding to the first element in the vector. If this fails, then we generate a slower, generic load instruction, which can load any data. This is likely to be very slow, as each vector element will be loaded separately and some logic will be used to combine all elements into a single SIMD vector.

Unused save elimination

The goal of this step is to minimize the overhead needed to save the SIMD vectors back into the scalar⁹ variables. The only purpose of the save phase is to make sure the SIMD generated code integrates smoothly with the non-SIMD code. To do so, the actual variables must be retrieved from the vector registers before they are used in non-SIMD code, or in SIMD load instructions.

A conservative solution is to generate save statements after each SIMD operations. Even though this is far from efficient in most situations, this ensures that the code is correct. Once the code is generated this way, use-def elimination can be applied to remove unneeded save statements.

Indeed, the purpose of this classical optimization is to remove statements that have no effects on the output of the program or application, and it therefore can easily detect that some save statements are not needed.

⁹Here, we mean scalar as opposed to vector, even though the variable can very well be an array

Chapter 5

SAC: SIMD Architecture Compiler

5.1 PIPS

PIPS, which stands for *Interprocedural Parallelization of Scientific Programs*¹, is a highly modular workbench for implementing and accessing various interprocedural compilers, optimizers, parallelizers, vectorizers, restructurers, and so on without having to build a new compiler from scratch. It has been used in various compilation areas since its inception in 1988, and further developed through several research projects.

The main goal of PIPS is to combine advanced interprocedural analysis and semantical analysis, while still being able to deal with real life problems in quite reasonable time. A multi-target parallelizer, PIPS incorporates many code analysis and transformations phases, as well as many options to tune its features. As of today, PIPS is a source-to-source Fortran translator. However, support for C is being added.

PipsMake

Internally, all the parsing, compiling, analysing, and transformation operations are split into basic, independent *phases*. Each of these phases perform a specific operation, like dependence graph generation or loop unrolling. To do this, each phase reads some resources generated by some other phase, and produces one or more new resources.

To let things run smoothly, and handle situations when some resources are made obsolete by the execution of some other phase, the phases are scheduled by PipsMake, an *à la make* mechanism that deals with resources and phases. The purpose of PipsMake is to automatically build the resources needed by a phase

¹In French, *Paralléliseur Inter-procédural de Programmes Scientifiques*

before starting it, and to rebuild the resources when they become obsolete.

To make this possible, all the data resources that are used, produced and transformed by a phase in PIPS are managed by another entity, PipsDBM. A phase can request some data structure in PipsDBM, use or modify them, create some other resources, and then store with PipsDBM the modified or created resources for later use. This data management facility allows PipsMake to know precisely when a resource has been modified, and thus perform its task efficiently.

Internal Representation

In PIPS, a Fortran program is represented as a tree, where each node represents an entity, control structure or any other paradigm of the program. This representation, called *internal representation*, or *IR*, is precise enough to allow the regeneration of the source code. Moreover, it is generic enough to be able to deal with other language than Fortran with little modifications. To sum up, this representation is a generalization of the classical expression tree, to deal with whole programs instead of simple expressions.

When started, the first thing PIPS does is to generate this IR from the actual source code. The code is split into modules, and a new resource is created to store the representation of each module. Thereafter, all phases interact with this representation only, and do not access the actual source files at all. Some special phases can also be used to generate new source files from the IR. We will now look at some important elements of the IR, as they will be needed to understand the details of SAC, and its implementation in PIPS.

In the IR, the **statement** is one of the most central elements: a **statement** represents a control structure or a “simple” line of code. Various variations are thus used to represent for loops, while loops, sequences, tests... The body of these structures is also a statement. A special kind of control structure, sequence, is used to actually represent the absence of control structure: it represents a list of statements that are executed sequentially. Finally, a **statement** can represent a single instruction: in this case, it represents a single line of code, which is in fact always a **call**.

A **call** is used to denote a function call. However, in PIPS, this is a bit more generic than just calling user functions. Indeed, this also includes calls to operators and other intrinsics of the language. Thus, an assignation $a = b$ is represented by a call to the intrinsic function² “assign operator,” with two arguments, a and b .

Each argument of a **call** is represented by an **expression**. As the name implies, it represents an expression, and can thus be a reference to a variable, or another **call**, thus allowing building complex expressions.

A **reference** is used to denote a variable access, which is in fact a memory

²PIPS defines a set of intrinsic functions, including all operators

access. The reference itself can be used for reading as well as writing. It may represent access to scalar variables as well as array variables, in which case the indices are represented by **expressions**.

Each function in the program is represented by a **module**. The whole program is thus a set of **module**. Each module is associated, among others, with a single **statement**, which represents the function body.

5.2 SAC overview

Once all those definitions are set, we can look at the way SAC is implemented into PIPS. SAC, the SIMD Architecture Compiler, tries to apply the techniques described in the previous chapter to generate SIMD code using the PIPS platform.

To achieve this goal, the process is split into a few phases. Some of the steps described earlier are grouped in a single phase, for performance reasons; others are implemented as their own phase. The choice was made based mainly on two considerations.

First, we tried to make a step, or part of a step into a phase whenever this was in fact more general, and could be used alone, in another context. Additionally, there are some situations when steps cannot be grouped together, when the second step needs resources that become obsolete due the first step. For example, if a phase needs the dependence graph, then it cannot be added to a phase that modifies the code, as the dependence graph would not be updated automatically. We need to have two separate phases in such situations, so that PipsMake can regenerate obsolete resources as needed.

Following those guidelines, SAC was implemented as a few phases, which are detailed in the next few sections:

- *SIMD Atomizer*: atomize the code as expected by the later phases;
- *SIMD Auto Unroll*: compute the “optimal” unroll factor, and unroll the code accordingly. The unrolling itself is done using an existing function, one the unroll factor has been computed as explained in section 4.3. Some additional built-in phases, partial evaluation and dead-code elimination, are then needed to cleanup the code;
- *Reduction Elimination*: minimize dependencies due to reductions, by using a reduction vector in place of a single reduction variable;
- *Single Assignment Form*: minimize dependencies due to the reuse of the variables;
- *Simdizer*: vectorize the code to take advantage of multimedia SIMD instructions. This phase also performs optimal load computations;

- *Use-def elimination*: perform use-def elimination to remove unneeded save statements. This phase is already implemented in PIPS, so we simply use it as is.

A configuration file is also used, which contains the opcode classes, the various opcodes that are associated, and the corresponding patterns. This file is used to setup the matching engine for the Simdizer phase, and for computing the optimal unroll factor with the more complex algorithm.

5.3 SIMD Atomizer

This phase implements an atomizer corresponding to what is described in section 4.2. PIPS already includes an atomizer³, which does not meet our need. Indeed, the built-in atomizer has a tendency to atomize all reference indices, thus making it almost impossible to detect adjacent memory accesses.

Fortunately, the included atomizer is in fact built as an atomizing engine, which uses callbacks to decide whether or not to atomize at some point. Therefore we only needed to provide the appropriate callbacks, letting the atomizer atomize anything but simple, two or three operands operations, and parameters of the form *reference + constant*.

One additional step is needed however. Indeed, this atomizer has a tendency to increase the complexity of the IR of the code. Indeed, each statement that is atomized is transformed into a sequence of statements. This makes it very difficult to identify anything in the later phases. To remedy this, we apply another built-in phase, *unspaghettify*, to “flatten” the IR, and cleanup the control structure.

5.4 Reduction Elimination

The implementation of this phase in PIPS is quite straightforward. First and foremost, the reduction detection algorithm is already implemented as a phase, which generates a resource containing the list of reductions. Each reduction that is returned corresponds in fact to a variable being reduced in a statement, by a specific operator. As a consequence, if a reduction is performed by multiple statements, multiple results will be returned. To use this, we simply need to let PipsMake know that we require this resource, and all will be taken care of automatically.

Our phase thus only performs the transformations needed to minimize the dependences and thus allows the efficient vectorization of the reduction, while the detection of those reductions is performed by a built-in phase.

³Actually, it includes two atomizers, but both display the same behavior

The implementation is quite simple and efficient. It consists on a simple traversal of the code, to search for the loops. Whenever a loop statement⁴ is identified, we traverse its body to see if some reductions are used. Note that this is performed in a bottom-up fashion, so that inner loops are processed before outer loops, for increased efficiency.

When handling the body, we perform two tasks. When a reduction is found, using the results from the reduction detection phase, we have to rename the variable, and to take note that this reduction is done in the loop, so that we can thereafter generate the prelude and compacting code.

To do so, we maintain a set of tuple associating a reduction, meaning a reduction operation and a reduction variable, to the number of time it is performed in the loop body and the vector variable that will be used to replace the reduction variable in the loop.

For each statement in the loop body, we look at the result from the reduction detection. For each of the reductions that have been detected, we first try to see if a similar reduction⁵ has already been detected. If this is the case, we increase the use count for this reduction, and get the associated vector variable. If not, we store a new association in our table of encountered reductions, with a use count of one, and create a new vector variable. We then replace all references to the reduction variable with a reference to the last element of the replacement vector variable.

When all statements in the loop body have been processed, we have a table containing the list of all reductions performed in loop, as well as the vector variable used to resolve the reduction, and the number of time this reduction is performed. Moreover, the loop body is already transformed adequately, with all references to the reduction variable transformed into references to elements of a reduction vector.

We can then easily generate the prelude code: this code sole purpose is to initialize each of the elements of the reduction vector, setting them to the neutral element for the reduction operation considered.

The compacting code can also be easily generated: it is just a matter of applying the reduction operation⁶ to all elements of the vector, in order to compute the actual result. It is important not to forget to include the original value of the reduction variable in the calculation.

Once this is done, we simply replace the loop statement with a sequence of statements: all the statements needed for the prelude, the loop statement itself, with its modified body, and the compacting phase. We must then apply once more the `unspaghettify` phase to make the control structure cleaner.

⁴There are a few kind of loops in PIPS: for, while... They are all similar enough for us to handle them the same way

⁵*Similar* meaning here that it performs the same reduction operation on the same variable.

⁶Which is, by definition, associative and commutative

5.5 Single Assignment Form

As explained in section 4.5, the single assignment transformation tries to minimize dependencies by ensuring, whenever possible, that a variable is set only once. This is done by renaming each variable, from the place it is written and in all the places which use this definition. If a variable is set more than once, then multiple names will be used to rename it, one per definition, so that each variable will be set only once in the resulting code.

This is done in two steps. First, we traverse the whole dependence tree to find out the number of incoming def-use arcs for each reference. This is used later in the transformation to make sure it is valid, as explained in section 4.5. At the end of this step, we have for each **reference** the number of def-use chains it terminates.

One may wonder why we do this calculation for each **reference**, instead of each tuple (*dependence graph node*⁷, *variable*). This is actually the same. Indeed, we perform our calculation for each **reference object**, and PIPS generates a new **reference** object for each access to the variable. Therefore, even if two references refer to the same thing, we will consider them different for this calculations, so we are in fact computing the expected information.

Once this is done, we perform the actual computation. This is done by looking sequentially at each node in the dependence graph. For each of the nodes, we then examine the def-use chains it generates, to build a list of the def-use chains that are subject to replacement. To make things simpler, this is built as a map, with the key being the reference set in the current node, and the items the list of references which depend on this one.

Each of these def-use chains consists of a *source* reference, being written, and a *sink* reference, being read. If we encounter a sink reference which has more than one incoming def-use arcs, as computed in the first step, then we rule out the replacement of all dependencies of the corresponding source reference. This ensures the transformation does not change the code, as explained in section 4.5.

Once we have examined all the conflicts, and built the list of all def-use chains that can safely be used for replacement, we can perform the actual variable renaming. For each source reference we create a new variable, and use it both in the source reference and the associated sink references. Once the whole dependence graph has been processed, the transformation is complete, and we can perform the actual vectorization.

5.6 Simdizer

This phase performs the actual vectorization process. This single phase performs all the operations described in sections 4.6 to 4.8, except the final use-def

⁷Each node corresponds in fact to a single statement.

elimination to remove unneeded save statements.

The whole process is performed with a single traversal of the function representation in IR. The representation is traversed in post-order, to search for statement sequences. Indeed, the vectorization unit is the sequence. Whenever one is encountered, the algorithm tries to vectorize it. It is important to notice that the vectorization of each sequence is performed independently, meaning the context is not taken into account.

This explains why we need to have a clean control structure, as else we may miss some vectorization opportunities. Although it may seem like a restriction to deal with sequences only, this makes actually things much simpler without sacrificing efficiency. Indeed, we designed our system so that it vectorizes sets of statements: this is why we unroll loops before, so that the loop body itself exhibits the parallelism. As a consequence, our technique has a slightly broader range than the classic loop vectorization techniques.

Whenever a sequence is found, all the statements that are part of the sequence are matched against the set of known patterns, and the results are stored in a map. This leads to better performance, as the matching is not done whenever we compare two statements.

Once all statements are matched, we use the greedy algorithm described in section 4.6 to group similar independent isomorph statements together. Anytime a statement is added to the group, it is moved at the exact end of the group. Eventually, the sequence of statements thus virtually becomes a sequence of groups⁸.

Whenever a group is found, the analysis described in section 4.7 is performed on the group, to generate a sequence of *SIMD statements*. These are not actual statements, but data structures used to describe SIMD statements, including the opcode to use and the arguments. This structure also contains the results of the vector dependence analysis, meaning that for each argument it points to all other SIMD vectors that share the same value at runtime.

By concatenating the sequences of SIMD statements generated for each group, we thus build a sequence of SIMD statements which performs the same calculations and operations as the original sequence of statements. Note however that this sequence may contain also non-SIMD statements: indeed, it can contain cells that describe *regular* statements, i.e. statements that we cannot or choose not to vectorize. This is particularly useful to handle non-matched statements, which are added to this list as is.

Once this list of SIMD statements is completed, it is used to generate the actual code, as described in section 4.8. The list is traversed in order, and for each SIMD statement a set of statements are generated, performing both the calculation and data transfers needed. Obviously, non-SIMD statements are generated as is, so that in the end they are actually copied from the original statements

⁸Virtually only, as the groups are not represented in the IR.

sequence to the new, optimized, statements sequence.

The load statements are optimized as described in the aforementioned section, using the vector dependence data; however, all transfers from the SIMD variables back to the actual variables are generated. Thus we finally get a new, optimized sequence of statements, which uses SIMD statements if possible, and which we can put in place of the original in the IR. When all sequences in the module have been processed, the transformation is complete.

Another phase, use-def elimination, is then used to eliminate those unneeded save statements, while keeping the ones that are needed. It is already included in PIPS, so we can simply use it, to obtain a huge improvement compared to the result of the previous phase, which generates one save statement for each generated SIMD instruction.

5.7 A full PIPS/SAC script

As we have seen, the vectorization of a function implies the execution of a few phases. Fortunately, PIPS include a scripting interface, which allows us to make a script to perform the full vectorization. An example of such a script is presented on figure 5.1. Some additional phases may also be used to output the new, optimized source file, in C or Fortran.

```
# Initialize PIPS
create b loop.f simd.f

# Atomize the code
apply SIMD_ATOMIZER
apply UNSPAGHETTIFY

# Optimized unroll
apply SIMDIZER_AUTO_UNROLL
apply PARTIAL_EVAL
apply SUPPRESS_DEAD_CODE

# Reductions elimination
apply CUMULATED_REDUCTIONS
apply SIMD_REMOVE_REDUCTIONS
apply UNSPAGHETTIFY

# Single assignment transformation
apply SINGLE_ASSIGNMENT

# Vectorization
apply SIMDIZER

# Unneeded save removal
apply USE_DEF_ELIMINATION

# PIPS cleanup
close
```

Figure 5.1: Sample vectorization script

Chapter 6

Experimental results

Unfortunately, our system is not complete enough to let us perform actual test on real applications. This is due to a few issues related to the platform we used, PIPS, and is in no way inherent to our design or algorithm.

Most important is the fact that PIPS deals mostly with Fortran, which means we cannot directly use usual benchmarks and test cases, which are mainly in C language. Thus some work would have been needed to convert them to Fortran 77, as supported by PIPS.

Moreover, using the Fortran code output prevents the generation of actual compiler intrinsics or inline assembly code, as compilers do not support those features. The C output allows those, yet PIPS does not allow us to control some important C details, like complex data types¹. This implies the C output still needs some rework, which is not performed now.

However limited these results are, our experiments on simple codes seem very promising. Indeed, tests operated on a few simple codes showed optimal results. The generated code is thus close or similar to hand-generated code, which is already quite promising. This is the case for vector additions and multiplications (figure 6.1), vector multadd (figure 6.2, and, to some extent, for the summation of all elements in a vector² (figure 6.3).

¹We mean complex as opposed to the simple data types supported by Fortran 77 and PIPS.

²Some additional operations need to be implemented for reduction handling to be optimal, but these modifications are quite simple: this is just a matter of propagating some information between the various phases, and of dealing correctly with it at each step.

<pre> PROGRAM LOOP INTEGER*1 I INTEGER*1 A(0:200) INTEGER*1 B(0:200) INTEGER*1 C(0:200) SUM = 0 DO 20 I = 0, 199 A(I) = B(I) + C(I) 20 ENDDO PRINT*,A(0) END </pre>	<pre> PROGRAM LOOP INTEGER*1 LU_INDO, LU_IBO, LU_NUBO, I INTEGER*1 A(0:200), B(0:200), C(0:200), &v8qi_vec0(0:7), v8qi_vec1(0:7), v8qi_vec2(0:7) 99998 CONTINUE DO 99997 LU_INDO = 0, 199, 8 CALL SIMD_LOAD8(v8qi_vec0, C, 0, LU_INDO) CALL SIMD_LOAD8(v8qi_vec1, B, 0, LU_INDO) CALL SIMD_ADD8(v8qi_vec2, v8qi_vec1, v8qi_vec0) CALL SIMD_SAVE8(v8qi_vec2, A, 0, LU_INDO) 99997 CONTINUE PRINT *, A(0) END </pre>
(a)	(b)

Figure 6.1: Vector addition, before (a) and after (b) SAC

<pre> PROGRAM LOOP INTEGER*1 I INTEGER*1 A(0:200) INTEGER*1 B(0:200) INTEGER*1 C(0:200) SUM = 0 DO 20 I = 0, 199 A(I) = B(I)*A(I) + C(I) 20 ENDDO PRINT*,A(0) END </pre>	<pre> PROGRAM LOOP INTEGER*1 I_7, I_6, I_5, I_4, I_3, I_2, I_1, &I_0, LU_INDO, LU_IBO, LU_NUBO, I INTEGER*1 A(0:200), B(0:200), C(0:200), &v8qi_vec0(0:7), v8qi_vec1(0:7), v8qi_vec2(0:7), &v8qi_vec3(0:7), v8qi_vec4(0:7), v8qi_vec5(0:7) 99998 CONTINUE DO 99997 LU_INDO = 0, 199, 8 CALL SIMD_LOAD8(v8qi_vec0, A, 0, LU_INDO) CALL SIMD_LOAD8(v8qi_vec1, B, 0, LU_INDO) CALL SIMD_MUL8(v8qi_vec2, v8qi_vec1, v8qi_vec0) CALL SIMD_LOAD8(v8qi_vec3, C, 0, LU_INDO) CALL SIMD_ADD8(v8qi_vec5, v8qi_vec2, v8qi_vec3) CALL SIMD_SAVE8(v8qi_vec5, A, 0, LU_INDO) 99997 CONTINUE PRINT *, A(0) END </pre>
(a)	(b)

Figure 6.2: Vector mult-add, before (a) and after (b) SAC

```

                PROGRAM LOOP
                INTEGER*1 LU_INDO, LU_IB0, LU_NUB0, I, SUM
                INTEGER*1 A(1:201), SUM_REDO(0:7), v8qi_vec0(0:7),
                &v8qi_vec1(0:7), v8qi_vec2(0:7)

                SUM = 0
99998 CONTINUE
                SUM_REDO(7) = 0
                SUM_REDO(6) = 0
                SUM_REDO(5) = 0
                SUM_REDO(4) = 0
                SUM_REDO(3) = 0
                SUM_REDO(2) = 0
                SUM_REDO(1) = 0
                SUM_REDO(0) = 0
                DO 99997 LU_INDO = 0, 199, 8
                    CALL SIMD_LOAD8(v8qi_vec0, A, 0, LU_INDO)
                    CALL SIMD_LOAD8(v8qi_vec1, SUM_REDO, 0, 0)
                    CALL SIMD_ADD8(v8qi_vec2, v8qi_vec1, v8qi_vec0)
                    CALL SIMD_SAVE8(v8qi_vec2, SUM_REDO, 0, 0)
99997 CONTINUE
                SUM = SUM_REDO(7)+SUM_REDO(6)+SUM_REDO(5)+
                &SUM_REDO(4)+SUM_REDO(3)+SUM_REDO(2)+
                &SUM_REDO(1)+SUM_REDO(0)+SUM

                PRINT *, SUM

                END

```

(a)

(b)

Figure 6.3: Vector sum, before (a) and after (b) SAC

Chapter 7

Perspectives

In just a few months, we could make a system to automatically vectorize code and generate multimedia SIMD instructions. Even though we were not able to validate our approach on actual code and benchmarks, initial results look very promising. Yet, there are many things that could be greatly enhanced, what could eventually imply better results in actual applications.

First and foremost, our algorithm can certainly be improved, on a variety of points. It is our belief that these modifications would improve the performance of real applications, although only actual tests can validate one approach or another. On the other hand, the simplicity of our algorithms implies that those improvements will most likely be more time consuming, which could be a problem if implemented in a commercial compiler.

One such target for improvement is the grouping policy. Indeed, we use a greedy algorithm, which works quite well in simple situations. This is more like an heuristic however, and a better, more precise, grouping policy could prove quite efficient. Various grouping alternatives may indeed be possible, and it may be interesting to investigate the various possibilities to choose the most promising one, maybe by also taking the available instructions into account.

Another idea would be to reorder the instructions in a group. Indeed, our algorithm first generates some groups of independent isomorphic statements, but eventually generates code for the statement in the exact order they appear in the original code. It may be interesting to reorder the statements in the groups, to makes the transfers of data to and from vector registers more efficient. This could indeed improve data locality, and help provide efficient memory access, as a vector would be filled from adjacent memory locations. Moreover, this may help improve vector reuse, if some statements can be reordered to make the vector identical to another, already loaded, vector.

Furthermore, our system does not support all instructions. Saturated arithmetic is not supported yet, and if a simple support can indeed be added, this would not detect all situations. We could easily modify the matching engine so that it can match tests, which would allow us to detect certain operations im-

mediately followed by a test as saturated arithmetic for example. However, for improved performance, saturated arithmetic should be detected more precisely, and used in the expression optimizer. Other non supported instructions are operations that perform some width conversion, as the operation thus induces a side-effect, or those that perform only partial operations, like multiplications¹.

In addition, some higher level optimizations may also be used to improve the generated code. One such optimization could be to first detect recurrent array access patterns, and thus generate code to reorganize the array before and after the actual processing. This way the processing can be optimized greatly by our existing algorithms.

¹Where one opcode may return the high bits of the return and the order the low order bits

Conclusion

SIMD multimedia extensions are very powerful, and can easily meet their goal of providing cheap performance for multimedia applications. Not only multimedia applications can benefit from this, but actually many more computation intensive applications. Multimedia extensions thus provide a cheaper alternative to the use of more powerful microprocessors.

The main drawback of this approach is that the application needs to be programmed specifically to benefit from this new feature. A few options are currently available, but all of them require spending time to optimize the application, and a deep knowledge of the target architecture. Right now, no production compiler is able to automatically generate optimized code using these SIMD multimedia extensions.

The goal of my internship was thus to add this capability to yet another compiler architecture, PIPS. There are already a few methods that have been quite appropriately discussed in the literature: vectorization, superword level parallelism extraction, and pattern matching. We have developed our own method, quite similar to superword level parallelism extraction, yet much simpler.

A drawback of the SIMD model used in general purpose processors is that some instructions need to be added to support packing and unpacking of the registers. Thus, the performance increase due to the parallelization of the calculations is negated to some extent by the overhead of packing and unpacking. To improve the performance, some advanced data rearrangement may be performed by the compiler. We tried to minimize the impact of this overhead in our system, by the mean of the vector dependence graph which allows us to reuse vectors or compute a vector from another one using special instructions from the instruction set.

To minimize this overhead even more, and thus improve the performance, more advanced transformations would also need to be performed. For example, to perform some operations on a subset of an array, one could first pack all relevant elements in a temporary array, perform the computation using SIMD instructions, and finally put the result back into the original array...

Another field that could be greatly improved is the use of the instruction set. Indeed, the operations provided are not the same as the regular instruction set: the multimedia SIMD instructions usually support saturated arithmetic, partial

multiplications, . . . This should be taken into account to optimize the expressions themselves, instead of simply vectorizing the code. Indeed, many multimedia applications would benefit greatly from saturated arithmetic, whereas most multimedia compilers today do not take full advantage of this.

Bibliography

- [1] R. J. Fisher and H. G. Dietz, “Compiling for SIMD within a register,” in *Languages and Compilers for Parallel Computing*, pp. 290–304, 1998.
- [2] R. Fisher, *General-Purpose SIMD Within A Register: Parallel Processing On Consumer Microprocessors*. PhD thesis, School of Electrical and Computer Engineering, Purdue University, 1997.
- [3] R. Lee, “Efficiency of microSIMD architectures and index-mapped data for media processors,” in *IS&T/SPIE Symposium on Electric Imaging: Media Processors 99*, 1999.
- [4] AMD, *3DNow! Technology Manual*, 1998.
- [5] AMD, *AMD Extensions to the 3DNow! and MMX Instruction Sets Manual*, 1999.
- [6] A. Peleg and U. Weiser, “MMX technology extension to the intel architecture,” *IEEE Micro*, vol. 16, no. 4, pp. 42–50, 1996.
- [7] Intel Corporation, “Instruction set progression from MMX technology through streaming SIMD extensions 2,” 2000.
- [8] H. Nguyen and L. K. John, “Exploiting SIMD parallelism in DSP and multi-media algorithms using the AltiVec technology,” in *International Conference on Supercomputing*, pp. 11–20, 1999.
- [9] R. Lee, “Subword parallelism with MAX-2,” *IEEE Micro*, vol. 16, no. 4, pp. 51–59, 1996.
- [10] MIPS Technologies, Inc, *MIPS Extension for Digital Media with 3D*, 1997.
- [11] P. Bannon and Y. Saito, “The Alpha 21164PC microprocessor,” in *IEEE Comcon '97*, 1997.
- [12] M. Tremblay, J. M. O’Connor, V. Narayanan, and L. He, “VIS speeds new media processing,” *IEEE Micro*, vol. 16, no. 4, pp. 10–20, 1996.

- [13] R. B. Lee, A. M. Fiskiran, and A. Bubshait, "Multimedia instructions in IA-64," in *IEEE International Conference on Multimedia and Expo*, 2001.
- [14] G. Erickson, "RISC for graphics: A survey and analysis of multimedia extended instruction set architectures," tech. rep., Department of Electrical Engineering, University of Minnesota, 1996.
- [15] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 363–400, 2000.
- [16] G. Cheong and M. Lam, "An optimizer for multimedia instruction sets," in *Second SUIF Compiler Workshop*, 1997.
- [17] A. Krall and S. Lelait, "Compilation techniques for multimedia processors," *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 347–361, 2000.
- [18] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *ACM SIGPLAN Notices*, vol. 35, no. 5, pp. 145–156, 2000.
- [19] M. Boekhold, I. Karkowski, and H. Corporaal, "Transforming and parallelizing ANSI C programs using pattern recognition," *Lecture Notes in Computer Science*, vol. 1593, pp. 673–84, 1999.
- [20] R. Manniesing, I. Karkowski, and H. Corporaal, "Evaluation of a potential for automatic SIMD parallelization of embedded applications," in *Fifth Annual Conf. of ASCI*, 1999.
- [21] R. Leupers, "Code selection for media processors with SIMD instructions," in *Design Automation & Test in Europe DATE*, pp. 4–8, 2000.
- [22] P. C. November, "Direct compilation of high level languages for multi-media instruction-sets," tech. rep., Department of Computer Science, University of Glasgow, 2000.
- [23] P. Jouvelot and B. Dehbonei, "A unified semantic approach for the vectorization and parallelization of generalized reductions," 1986.