Exploiting SIMD Parallelism with the CGiS Compiler Framework

Nicolas Fritz*, Philipp Lucas*, Reinhard Wilhelm

Universität des Saarlandes, 66041 Saarbrücken, Germany
`[cage|phlucas|wilhelm]@cs.uni-sb.de`

Abstract. Today's desktop PCs feature a variety of parallel processing units. Developing applications that exploit this parallelism is a demanding task, and a programmer has to obtain detailed knowledge about the hardware for efficient implementation. CGiS is a data-parallel programming language providing a unified abstraction for two parallel processing units: graphics processing units (GPUs) and the vector processing units of CPUs. The CGiS compiler framework fully virtualizes the differences in capability and accessibility by mapping an abstract data-parallel programming model on those targets. The applicability of CGiS for GPUs has been shown in previous work; this work presents the extension of the framework for SIMD instruction sets of CPUs. We show how to overcome the obstacles in mapping the abstract programming model of CGiS to the SIMD hardware. Our experimental results underline the viability of this approach: Real-world applications can be implemented easily with CGiS and result in efficient code.

1 Introduction

Recent hardware development is leading from traditional core frequency increase towards parallelism [3]. Even standard PCs feature parallelism on several levels of granularity. Multiprocessor systems support a MPMD model, which distributes tasks to different cores. GPUs (graphics processing units) [15] and SIMD units of CPUs follow the SPMD paradigm. Exploiting this parallelism, however, is not sufficiently supported by common programming languages, which are still tightly coupled to the sequential computing model. Algorithms using SIMD instructions are commonly written in assembly language or low level programming language extensions (intrinsics) [22].

The CGiS system strives to open up the parallel programming capabilities of commodity hardware to ordinary programmers. It raises the abstraction level high enough, so that the developer is kept away from all hardware intricacies. A CGiS program consists of parallel forall-loops iterating over streams of data and sequential kernels called from those loops. The CGiS compiler framework supports both CPUs and GPUs as targets, exploiting their characteristics automatically. For GPUs this has been presented in [9]. The paper in hand focuses on

* In part supported by DFG grant WI576/10.

the SIMD back-end of the CGiS compiler generating code for Freescale's AltiVec and Intel's SSE, and presents a number of transformations and optimizations.

Modern GPUs offer hundreds of floating point units, which can work in a SIMD fashion on vectorial values or on any kind of scalar data [14]. Thus, GPUs can even execute scalar operations in parallel, offering heterogenous parallelism. In contrast to that, the SIMD units of PowerPCs and various generations of Intel Pentiums have only up to three 4-way SIMD processing units. This means that GPUs offer both SIMD parallelism in a single element and across a multitude of elements, whereas only the element-wise parallelism is exploitable by SIMD CPUs.

CGiS offers two levels of explicit parallelism, large scale SPMD parallelism by the iteration over streams and small scale SIMD parallelism by vectorial data types. A CGiS back-end needs to map these parallelisms to the ones offered by the target architecture. For GPUs this is a one-to-one mapping; for SIMD CPU architectures, the back-end has to chose which parallelism opportunity to map to the hardware features.

A method to map SPMD parallelism to the SIMD hardware is *kernel flattening*. This operation breaks down compound data into scalars to enable sensible packing of new vectors for parallel execution. To ensure the preservation of the program semantics, static program analyzes are used to guarantee the premises. This also requires automatic reordering of the input data which can be done locally to the routine or globally for all routines.

In many algorithms memory accesses dominate the computations. This makes the overall performance dependent on the memory connection. Hardware developers incorporate caches to speed up the access, but computations on large data sets make evictions inevitable. To make best use of the caches, a mechanism for loop sectioning is integrated in our SIMD back-end. The iteration of the data streams is adapted to the cache size and the stream layout.

The remainder of this paper is organized as follows. Section 2 gives a short overview of the current SIMD instruction sets and Section 3 provides a more in-depth look on CGiS, comparing it to related work. The SIMD back-end and its optimizations are set forth in Section 4, and examples and experimental results are presented in Section 5. Future work is discussed in Section 6, and Section 7 concludes the paper.

2 Hardware

The first SIMD instruction set in commercially successful desktop processors, the *Multimedia Extensions (MMX)* [11], was introduced by Intel in 1997. MMX extended the core instruction architecture with eight 64-bit registers and provided only integer instructions. It was followed by the Streaming SIMD Extensions (SSE) [7] in 1999. The first version of SSE provided eight 128-bit registers and a set of floating-point instructions. The SSE instruction set was successively extended by introducing integer support and horizontal operations (SSE2 and

SSE3). SSE4 promises a broader connection to the SIMD processing unit as well as more horizontal instructions to speed up common algorithms.

The PowerPC architecture was augmented with the Velocity Engine or *AltiVec* [5, 6] in 1999. It provides thirty-two 128-bit registers to hold vectors and supports integers of various widths as well as floating-point data. In contrast to SSE, AltiVec supports very powerful data reordering or permutation instructions, allowing arbitrary interchange of input vectors.

3 The CGiS Framework

CGiS [9, 10] is a data-parallel language for GPUs and CPUs. The CGiS language and the runtime system abstract the target in a uniform way. In particular, it is invisible to the programmer on which target the generated code is executed. CGiS is not intended to replace a programming language for a complete application. Instead, a data-parallel algorithm can be expressed in CGiS and then called as a simple subprogram of an application.

3.1 CGiS

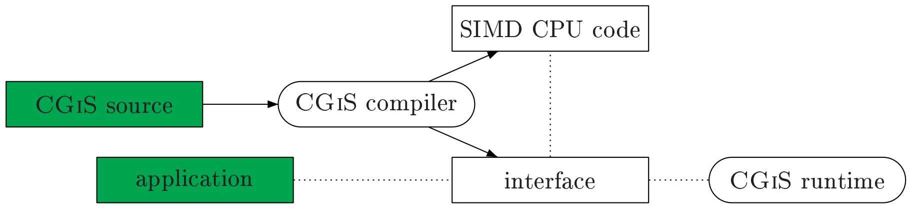


Fig. 1. Using CGiS. Arrows denote in- and output, dotted lines denote linkage. The filled rectangular nodes are user supplied code. The oval nodes are part of CGiS, and the other code components are generated by the CGiS compiler.

Figure 1 shows the usage pattern of CGiS. A source code file is fed to the compiler, which outputs code for the desired target (here: SIMD CPU code) and code for interfacing with the main application. The programmer interacts only with this interface code in a uniform way.

For an example of a CGiS program, see the code in Figure 2. It presents the rc5 cipher encryption [19] with a static number of 31 rounds. CGiS files are divided into three sections. An `INTERFACE` section defines the global data of the program; in this case, a one-dimensional stream of unspecified size of integer tuples (the stream to be encrypted) and a field of 32 integer key pairs. A `CODE` section defines the *kernels* operating in parallel on elements of *streams*. Here, the procedure `rc5` operates on stream element `AB`. It is an `inout` parameter, meaning it is read and written in the same iteration. The second parameter `S` is a reference to a stream, denoted by `<_>`. The called procedure `load` looks up

```
PROGRAM rc5_encryption;

INTERFACE
extern in uint2 S<32>;
extern inout uint2 ABs<SIZE>; // The stream to be encrypted.

CODE
procedure encrypt(inout uint2 AB, in uint2 S<_>)
{
  uint2 S01; uint i = 0;
  load(S,i,S01); // Get key-pair at 0.
  uint A = AB.x+S01.x, B = AB.y+S01.y;
  while(i<31) {
    i = i + 1;
    load(S,i,S01);  // Get key-pair at i.
    A = ((A^B)<<<B) + S01.x; // <<< is a left-
    B = ((B^A)<<<A) + S01.y; //     rotation.
  }
  AB.x = A; AB.y = B;
}

CONTROL
forall(AB in ABs) encrypt(AB,S);
```

Fig. 2. CGiS encryption of a stream with 31 rounds of rc5. A stream element consists of a pair of unsigned integers.

the `i`-th element of `S` and stores it in `S01`. The suffixes `.x` and `.y` on vectorial values denote component-selection: A vectorial value with a size of at most four can be treated as a structure with components `x`, `y`, `z`, `w`. The `CONTROL` section initiates a computation on streams. In this case, the kernel `rc5` is invoked for the elements of the stream `ABs`. The computations on the elements get scheduled in parallel (SPMD).

CGiS features a relatively standard, imperative programming language to describe the kernels in the `CODE` section. It is based on C, but lacks pointers: Arrays are always accessed with indices, and function outputs are implemented with pass-by-value-result parameters. These restrictions are a consequence of CGiS' ancestry as a GPU programming language. Also stemming from this are the native vectorial types and operations, special instructions for reordering components and guarded executions. Element types are single-precision floating point or signed or unsigned integer.

Streams can be accessed through read-write iterators, with relative and absolute read accesses, and absolute write accesses. The kernels are scheduled by a simple language featuring sequential specification of parallel executions in the `CONTROL` section. The runtime system is responsible for synchronization and sequencing of memory accesses to ensure a well-defined semantics. The `INTERFACE` section declares the interface to other CGiS programs and to the application: The application passes pointers to the input data and receives the output data through C functions generated for the interface code. The target remains hidden in this approach: The main application uses the generated code as a black-box, consuming streams of input data and producing streams of output data.

CGiS was originally deceived as a language for general-purpose computations on GPUs [10, 15]. As such, much of its syntax and semantics are owed to the hardware peculiarities of GPUs. SIMD CPUs can also make use of floating point vectors, but they lack the abundance of execution units. Therefore, a translation based on the same kind of parallelism available on GPUs is bound to produce lackluster results. From the two levels of parallelism mentioned above, SIMD parallelism on vectors and SPMD parallelism on stream elements, CPUs lack the GPU's large parallelism of the second kind. Section 4 shows that, with appropriate transformation on the source code inside the compiler, data-parallel algorithms expressed in CGiS can nevertheless also efficiently be executed on SIMD CPUs.

3.2 Related Work

Exploiting SIMD parallelism from standard C code is a complicated task. Common C compilers like gcc or icc are facing a multitude of problems both in analyzing the input code and in mapping it efficiently to the restricted SIMD hardware; many algorithms are still implemented by hand in assembly code or intrinsics, or using prefabricated libraries [17, 22]. CGiS features a stream programming model, avoiding some of these problems and offering new opportunities to overcome others. The expressibility is restricted with respect to the full possibilities of C code, but it allows easier exploitation of parallelism.

The CGiS SIMD back-end shares a set of common problems with other SIMD code generation approaches. One of the major problems is data alignment, because SIMD hardware usually is limited to accessing 16-byte aligned addresses [16]. Because CGiS operates solely on non-overlapping arrays (streams) with indexed accesses, alignment analysis becomes easier and permutation operations can be kept local. Also control flow prevents parallelization, and for SIMD traditional control flow conversions have to be employed [1, 23].

Other problems are avoided by language design or have to be tackled differently. As explicit data parallelism is mandatory for CGiS programs, extensive data-dependency analyses are obsolete. Specialized operations such as saturated operators or bit rotation operation are common to multimedia applications. These operations have to be reconstructed from C code by idiom recognition [17, 18], whereas they are present in CGiS. To utilize SIMD potential on scalar code, superword level parallelism is able to recognize isomorphic operations on sequential, scalar code [8, 21]. CGiS offers small (up to four components) vectorial types and componentwise operations, enabling the programmer to express isomorphic operations in their natural form. Exploiting SIMD parallelism from scalar code [13] is handled by cross-kernel-parallelism due to a transformation called kernel flattening. Conversion between element types of different length is a severe problem in C based approaches [22]; in CGiS, all data types are 32-Bit long.

4 The SIMD Back-end

This section deals with the transformations and optimizations which are necessary for the SIMD back-end of the CGiS compiler.

The challenges in generating efficient SIMD code differ from the ones in generating GPU code. Increased performance compared to scalar execution can only be achieved by exploiting vector parallelism. Each vector register of the supported SIMD hardware is 128 bit wide and can contain 4 floating-point or 4 (signed or unsigned) integer values. Mapping the stream computation to this hardware is hindered by the following issues:

– *Misalignment and data layout.* CGiS allows streams of arbitrary data elements allocated by standard allocation functions in the application. Because data can only be accessed with 16-byte aligned loads[1], in general, data must be reordered at some point. Consider the example in Figure 2 which describes a CGiS function encrypting a stream of pairs of unsigned integers with the rc5 encryption algorithm. With two integers per element, every odd element is not aligned for SIMD hardware.[2]
  The stream elements are compounds and the operations work on single components. There is no efficient SIMD exploit when processing one or two stream elements at a time as computations on the components are not uniform. Neither the data layout nor the alignment of the tuples match the requirements for SIMD vectorization.
– *Gathering operations.* Accesses to the main memory of a CPU are inherently slow. Thus, on-chip caches are employed to speed up the access to re-used data. Apart from arbitrary stream *lookups*, the CGiS language allows stream element loads or *gathers* relative to the element currently processed (neighborhood operations). Depending on the organization of the stream data and the shape of the accessed neighborhood, the CGiS compiler can adapt the stream iterations to increase cache performance.
– *Control flow.* Vectorizing code with control flow structures requires code transformation to ensure each of the stream elements processed in parallel enters the correct control flow branch. As in traditional vectorization, this is done by if- and loop-conversion and inlining.

4.1 Kernel Flattening

The main challenge in generating efficient SIMD code is data arrangement and meeting the alignment requirements of data accesses. A solution to this alignment and data layout problem is *kernel flattening*.

[1] The unaligned loads supported by SSE2 severely impact execution time.

[2] We assume that at least the first element of every stream processed is 16-byte aligned. Another example of a stride-one stream where not every stream element is aligned is the YUV-stream depicted in Figure 3.

Kernel flattening is a code transformation on the intermediate language. It processes a single kernel and splits all stream elements and variables into scalar variables. This also includes operations on those variables: Every operation is copied and executed on each former component of the variable. Figure 3 shows the flattening operations applied to a simple CGiS procedure, transforming YUV color values into RGB values. The parameters `YUV` and `RGB` and the constant vectors are split into 3 scalar variables each. The assignment to `RGB` and the computations are split as well.

```
procedure yuv2rgb(in float3 YUV, out float3 RGB)
{
  RGB = YUV.x + [ 0, 0.344, 1.77 ] * YUV.y + [ 1.403, 0.714, 0] * YUV.z;
}

procedure yuv2rgb_f (in  float YUV_x, in  float YUV_y, in  float YUV_z,
                     out float RGB_x, out float RGB_y, out float RGB_z)
{
  float cy = 0.344, cz = 1.77, dx = 1.403, dy = 0.714;
  RGB_x = YUV_x +              dx * YUV_z;
  RGB_y = YUV_x + cy * YUV_y + dy * YUV_z;
  RGB_z = YUV_x + cz * YUV_y;
}
```

Fig. 3. The procedure `yuv2rgb` transforms YUV color values into RGB. As in common GPU languages, scalar operands are replicated to match the number of components of the target or operation. `yuv2rgb_f` is the result of the flattening transformation applied to `yuv2rgb`. Each component becomes a single scalar variable or parameter and all vector operations are replaced by scalar ones.

The procedure resulting from kernel flattening can be executed in parallel. After compound variables have been broken down to scalar ones, these can be subjected to SIMD vectorization. Four consecutive elements for each scalar variable stream can now be loaded into one vector register, and immediate constants are replicated into a vector. Because the original data elements of the stream are possibly ordered in tuples (e. g., the YUV-stream in Figure 3), data has to be reordered during execution or beforehand. The SIMD back-end supports local and global data reordering depending on the re-usability of the reordered data. Whereas global reordering is basically a reordering in memory, local reordering inserts code that reorders these elements in registers at the beginning of the function and at the end. For the previous example the possible stream access patterns are shown in Figure 4. Sequential execution accesses one YUV-triple per iteration. Global reordering splits the YUV-stream into three streams. Thus, in each iteration, four elements of each former component can be loaded into a vector register and processed. Local reordering takes the stream as it is and inserts permutation operations at the start and the end of the flattened procedure.

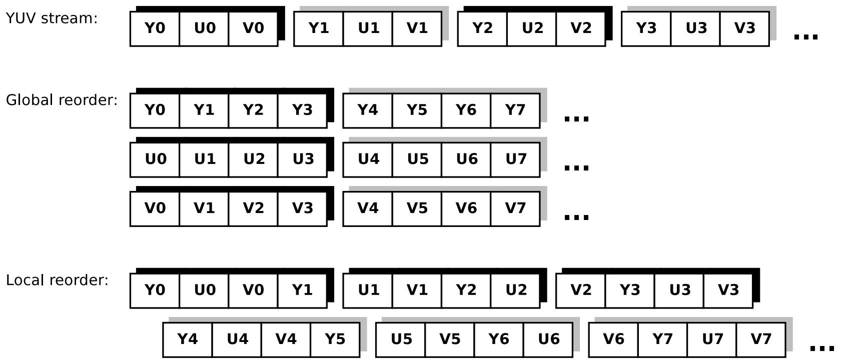

YUV stream:
Y0 U0 V0 Y1 U1 V1 Y2 U2 V2 Y3 U3 V3 ...
Global reorder:
Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7 ...
U0 U1 U2 U3 U4 U5 U6 U7 ...
V0 V1 V2 V3 V4 V5 V6 V7 ...
Local reorder:
Y0 U0 V0 Y1 U1 V1 Y2 U2 V2 Y3 U3 V3
Y4 U4 V4 Y5 U5 V5 Y6 U6 V6 Y7 U7 V7 ...

Fig. 4. Streams can be reordered globally or locally. Global reordering copies the data in memory before and/or after execution, depending on the data flow of the stream. Local reordering uses SIMD permutations. The layout of the data in memory remains unchanged. Different shades denote consecutively accessed data per iteration.

Per default, the CGiS back-end uses local reordering, but the programmer can force global reordering by annotations. Global data reordering requires input stream data to be loaded before and output stream data to be stored after execution. Thus, the higher reordering costs with respect to local reordering are amortized only if the reordered stream is processed several times with gathers and lookups.

Lookup and gather operations are split as well. In case of global reordering, the gathers and lookups are straightforward, because only alignment has to be taken care of. As for local reordering, on gather operations the loaded and reorganized data can be kept minimal as the offset from the current element is statically known. On the other hand, data-dependent lookups result in four different scalar loads and the reconstruction of a vector. With too many of these lookups the benefit of vectorizing the function might get negated.

For the rc5 encryption, this means that the `inout` parameter `AB` is split into an `inout` parameter `AB_x` and an `inout` parameter `AB_y`. All operations are made scalar enabling SLP execution. Data reordering instructions are inserted allowing stride-one access to `AB_x` and `AB_y`. From the perspective of data layout and alignment, four elements can be processed in parallel. The lookup function `load` only depends on the scalar `i`. With data-flow analysis it can be determined that `i` is constant across all elements processed in parallel. CGiS allows the user to annotate uniform variables to guide the compiler. Each of the parallely processed elements wants to load the same value. So the desired vector can be reconstructed from one SIMD load and one or two permutations.

4.2 Loop Sectioning

Many data parallel algorithms, especially in image processing, require the gathering of nearby data elements. One example for such an image processing algorithm

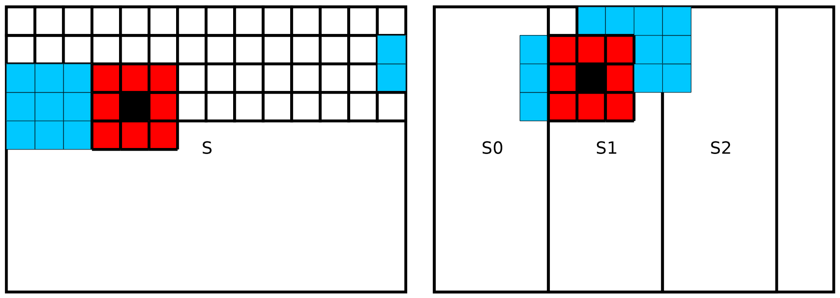


Fig. 5. The CGiS compiler can adapt the iteration order to increase cache hit rates in gather operations. Blue (light) squares are cached elements, the black square is the currently processed element and the red (dark) ones are the neighborhood accessed in this iteration. The left picture shows row-by-row iteration. In the right picture stripe-by-stripe is processed, showing the benefit of additional overlapping of cached data.

is the *Gaussian blur* described in Section 5.2. CGiS supports gathering operations that let the programmer access stream elements relative to the current position in the stream. When iterating over a two-dimensional stream column-by-column or row-by-row, it is possible that data elements already loaded and present in the data cache are evicted and have to be loaded again. To make best use of the cached data, the CGiS compiler can adapt the iterations over the stream dividing the field into smaller stripes that better match the cache size and organization of the processor. This optimization was inspired by [2]. With the smaller width of the stripes, there is more overlapping in-between row iteration.

This is possible only on architectures which allow a direct control of the iteration order. As for example GPUs do not allow this detailed iteration control, the GPU back-end of the CGiS compiler cannot make use of this optimization.

As an example, consider Figure 5. A two-dimensional field S is processed, and for each element its 8 immediate neighbors are gathered. The blue (light) squares are data elements that have been loaded in former iterations and are thus present in the data cache. The currently iterated element is colored black, and the gathered elements are red (dark). In the left part, the iteration sequence is simply row-by-row. The right part shows the same field subdivided into smaller stripes S_i. Each S_i is also processed row-wise. But with the reduced row width, the cache hit rate is increased, because there is still data present in the cache from the last processed row.

The size of the stripes is determined by the cache size and the memory requirements as follows. To determine the dimension in which the stripes run, we investigate the access pattern of the gather operations. The dimension which gives rise to the most data accesses defines the run direction. We assume that the two-dimensional stream is stored row-wise in memory. The stripes then run column-wise. Iteration is row-wise inside the stripes. For each stream, o determines the maximum of iteration lines or rows crossed by the access pattern, e. g., in Figure 5 o is 3. o is the sum of maximum absolute offsets in stripe direction plus one for the current line. For a given parallel kernel-execution k, the CGiS compiler decides the width of the stripes S_k from the cache size C, the cache line size l and the size of the stream elements read and written. (Different architectures with different cache sizes are selected at compile-time.) δ is a constant number that represents the local data that is needed in each iteration such as intermediates and other stack data. Assume that k accesses stream elements with an element size of a_i and the gathers for a_i cross o_i lines. These parameters are statically known and result in a simple heuristic for computing the stripe width:

$$S_k = \left\lfloor (C - \delta)/(\sum o_i \cdot a_i) \right\rfloor_l .$$

$\lfloor\ \rfloor_l$ rounds down to the nearest multiple of l. S_k does not need to be constant across a whole program but is adapted to each specific kernel execution.

Should the size of the field not match a multiple of the stripe width, the remaining elements are processed by normal iteration.

4.3 Control Flow Conversion

The three main control flow constructs of CGiS are procedure calls, conditionals and loops. Breaks are represented as modifying the loop control variable, so that each loop has exactly one exit. All transformations of the control flow conversion are executed on the intermediate representation of the CGiS program.

By default, calls are fully inlined in the SIMD back-end, although it is possible to force separate functions. We found that generating true calls increases the runtime of the application. Most parameters are present in vector registers, and passing those as arguments induces additional stores and loads.

If-conversion is the traditional way to convert control-dependencies into data-dependencies. A *mask* is generated for the condition. The execution of each statement in the conditional body is guarded by that mask [23]. In CGiS, the masks are the results of vector compare operations. These component-wise operations yield a vector that contains all 0s at an element if the comparison failed for that element, all 1s otherwise. Because current SIMD hardware does not feature guarded assignments, the Allen-Kennedy algorithm of [23] has to be adapted in the following way.

Let I be a basic block containing an if-statement with condition C_I and its associated mask M_I. For simplicity, we consider only a simple conditional body, with one block T_I in the `true`-branch and one block F_I in the `false`-branch. The control flow join is denoted J_I. Let L_I be the set of variables live at J_I, W_T the set of variables written in T_I and W_F is the set of variables written in F_I. The algorithm which inserts the additional operations required for the if-conversion is given in pseudo-code in Figure 6.

During the if-conversion phase, for each I the sets S_T and S_F are determined. For each control flow branch, copies of the variables written and live after the branch are inserted at the beginning of the respective branch. After the end of a branch, select instructions (like ϕ-functions from SSA [12]) are inserted which select the new value for the written variable depending on the generated mask.

```
Use live variables analysis to determine $L_I$
Use reaching definitions analysis to determine $W_T$ and $W_F$
Build intersections $S_T = W_T \cap L_I$ and $S_F = W_F \cap L_I$
Foreach $v_T \in S_T$
  insert ${v_T}' = v_T$ at the beginning of $T_I$
  insert $\text{select}(v_T, {v_T}', M_I)$ at the end of $T_I$
Foreach $v_F \in S_F$
  insert ${v_F}' = v_F$ at the beginning of $F_I$
  insert $\text{select}({v_F}', v_F, M_I)$ at the end of $F_I$
```

Fig. 6. Pseudo code for additional insertion of copies and select operations used in if-conversion.

The conversion of loops is pretty straight forward. For the loop condition, a mask is generated as well, and the loop is iterated as long as the mask is not completely 0 (signifying that *all* elements or the SIMD-tuple have finished iteration). If the mask is completely 0, then the loop can be exited.

Conversion of nested control flow statements is also supported. When the mask of a condition is generated for a nested statement, it is always combined with the mask of the control flow statement via binary `and`.

5 Examples and Evaluation

Three data parallel algorithms from different application domains will demonstrate the fitness of the SIMD back-end to the CGiS framework. Though all naturally parallel, each of those algorithms requires different optimization to run efficiently. Thus, they serve as representatives for larger categories of similar applications.

The two target platforms were a Freescale PowerPC G5, 1.8 GHz, running under Mac OSX, and an Intel Core 2 Duo 1.83 GHz running under Linux. The generated intrinsics code was compiled with gcc 4.0.1. Figure 7 shows the aggregation of the experiments for both hardware platforms. Speedup factors for SIMD only differ in the rc5 example as the SSE hardware does not support rotates or register-dependent shifts.

5.1 rc5 Encryption

rc5 [19] is a block cipher encryption that works on a stream of integer tuples. Each tuple gets modified by rotating and binary xor over a certain number of rounds. A parallel implementation of rc5 encryption requires the data to be reordered. Global data reordering via memory copy is not a valid option as it increases overall computation time drastically and does not amortize by the gain in computation speed. The alternative is local data reordering and is a automatically done by kernel flattening. For the tests, the message length to be encrypted is between 64k and 320k integers.

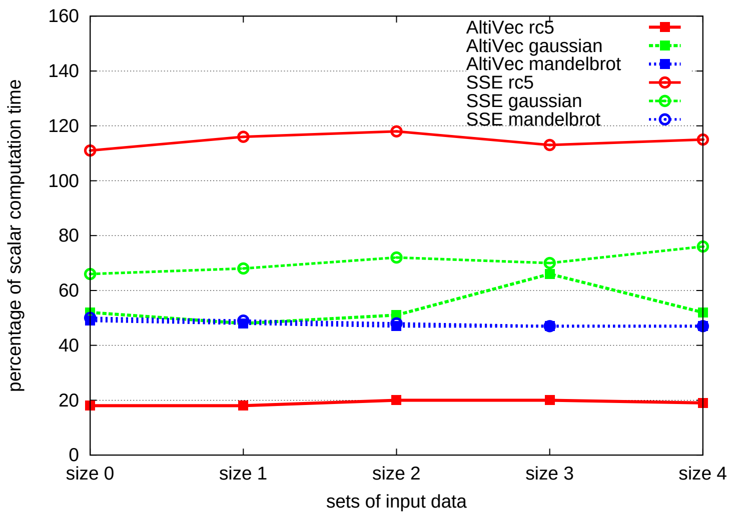


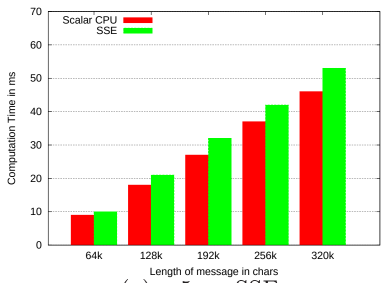
Fig. 7. Execution times of AltiVec and SSE hardware relative to scalar execution of PowerPC and Core 2 Duo. While Mandelbrot and Gaussian blur perform equally well on both architectures, rc5 is significantly slower on the Core 2 Duo due to the missing hardware rotation.

Figure 8.a holds the results of the generated SSE code. Because the Streaming SIMD Extensions do not support vector register dependent shifts or rotates, these operations must be done by the ALU, forcing data to go through memory twice. This has a severe impact on the computation time. The average increase of SIMD with respect to scalar code is about 20%. The results of the AltiVec implementation are shown in Figure 8.b. For all input sets the speedup is about 400%. As gcc did not recognize the shift patterns as rotates, it did not use the scalar rotate of the PowerPC, decreasing the scalar performance.

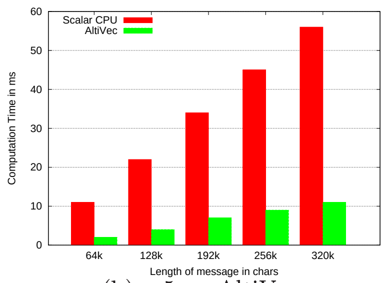
5.2 Gaussian Blur

Gaussian blur is an image processing algorithm to produce a blurring effect. For each pixel, its color and the colors of its neighbors are weighted and combined into a new color. Here the memory accesses strongly dominate computations. Our image data is stored in RGBA format, of which only the RGB values are considered. To increase performance in the gathering operations, cache sensitive iteration tries to make best use of the data already present in the cache. For the tests, an input image has been scaled, doubling the image size per test case.

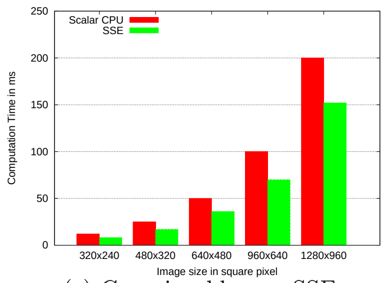
The SSE results in Figure 8.c do not show large improvement over the scalar implementation. The increased memory accesses together with the weak memory connection of the SSE unit thwart any performance gain by the parallel execution. The execution times on AltiVec hardware in Figure 8.d show an improvement of roughly 50% and scale well with the size of the inputs.



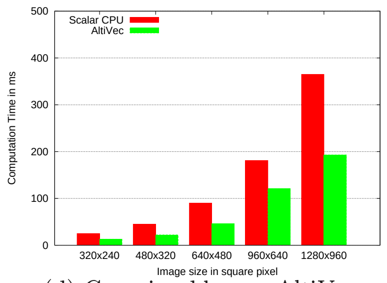
(a) rc5 on SSE



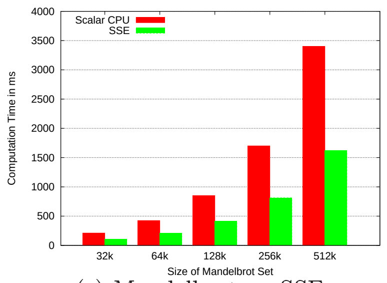
(b) rc5 on AltiVec



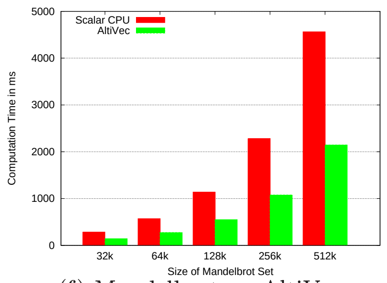
(c) Gaussian blur on SSE



(d) Gaussian blur on AltiVec



(e) Mandelbrot on SSE



(f) Mandelbrot on AltiVec

Fig. 8. Performance evaluation of three test sets on SIMD CPUs. The left column shows the computation times on SSE, the right one the computation times on AltiVec.

5.3 Mandelbrot Set

Computing the Mandelbrot set is a well-known, computationally heavy algorithm. For a point $z \in \mathbb{C}$ in the complex plane, the sequence $z_0 = z$, $z_n = z_{n-1}^2 + z$ is computed until $|z_n^2| \geqslant 2$ for some n or a maximal iteration count n' is reached. Afterwards, the final iteration count n is mapped onto a color.

Parallelization of this algorithm is only possible with control flow conversion. For this example, both SSE and AltiVec implementation show speedups of factor 2 (Figure 8.e–f) across all inputs. While the SSE instruction set offers the possibility to read the results of a compare directly to scalar hardware, for AltiVec writing of control register bits must be enabled and conditional jumps depending on those control bits are introduced.

6 Future Work

The SIMD back-end of the CGiS compiler is still under development. The main focus up to this point was to generate efficient code, i. e., faster than scalar code, for suitable applications. The next goal is the refinement of the existing optimizations. More program analyzes and better heuristics should replace the current heuristics.

Although intrinsics are a comfortable way of generating SIMD code, they have limitations. The conditional jumps using the control register of the PowerPC have to be inserted via inline assembly resulting in inefficient code. Furthermore, with pure assembly code emittance the compiler has more control over register allocation, which is imperative on the SSE architecture. This also enables the optimization of register caching for gather operations [20]. Compiling directly to assembly would offer also easy access to other processor features. For example, conditionals in loops can be optimized by introducing flags to avoid the generation of the masks for the if-statement, should there be no else-branch associated with the if. Also, we plan to extend the compiler to the Cell processor, which offers parallelism on several kinds [4]. We believe that the CGiS model can efficiently be mapped to the parallelisms allowed by the Cell processor.

7 Conclusion

This paper presents the SIMD back-end of the CGiS compiler framework in its current state. Generating efficient SIMD code for data parallel algorithms is a demanding task as many restrictions like data layout, control dependencies and other characteristics of the hardware avoid vectorization.

We introduce the program transformation of kernel flattening combined with local data reordering to solve the problem of data layouts that are not suitable for stream processing otherwise. On memory dominated algorithms, we try to increase performance by making best use of data caches by adapting the iteration sequence. Control flow is straightened by full if- and loop-conversion offering the possibility to parallelize functions with control dependencies. The experimental results show the viability of this approach. Though the number of examples is not exhaustive, the applications each stand for a whole category of similar applications in the field of encryption, image processing and mathematical calculations.

References

1. Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.
2. Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of PLDI*, pages 279–290, 1995.
3. David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1999.

4. Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for a cell processor. In *Proceedings of PACT*, 2005.
5. Freescale. *AltiVec Technology Programming Interface Manual*, June 1999. ALTIVECPIM/D 06/1999 Rev. 0.
6. Freescale. *AltiVec Technology Programming Environments Manual*, April 2006. ALTIVECPEM/D 04/2006 Rev. 3.
7. Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, May 2007.
8. Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. Technical Report LCS-TM-601, MIT Laboratory for Computer Science, November 1999.
9. Philipp Lucas, Nicolas Fritz, and Reinhard Wilhelm. The CGiS compiler—a tool demonstration. In *Proceedings of the 15th International Conference on Compiler Construction (CC'06)*, volume 3923 of *LNCS*, pages 105–108, 2006.
10. Philipp Lucas, Nicolas Fritz, and Reinhard Wilhelm. The development of the data-parallel GPU programming language CGiS. In *Proceedings of ICCS*, volume 3994 of *LNCS*, pages 200–203, 2006.
11. Millind Mittal, Alex Peleg, and Uri Weiser. MMX technology architecture overview. *Intel Technology Journal*, Q3:12, 1997.
12. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
13. Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data of simd. In *Proceedings of PLDI*, 2006.
14. NVIDIA. *CUDA Programming Guide Version 0.8*, February 2007.
15. John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.
16. Ivan Pryanishnikov, Andreas Krall, and R. Nigel Horspool. Compiler optimizations for processors with SIMD instructions. *Software—Practice & Experience*, 37(1):93–113, 2007.
17. Gang Ren, Peng Wu, and David Padua. An empirical study on the vectorization of multimedia applications for multimedia extensions. In *IPDPS*, 2005.
18. Gang Ren, Peng Wu, and David A. Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *Proceedings of LCPC'03*, pages 420–435, 2003.
19. Ronald L. Rivest. The RC5 encryption algorithm. In *Practical Cryptography for Data Internetworks.* IEEE Computer Society Press, 1996.
20. Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *Proceedings of PACT*, pages 45–55, 2002.
21. Christian Tenllado, Luis Piñuel, Manuel Prieto, and Francky Catthoor. Pack transposition: Enhancing superword level parallelism exploitation. In *Proceedings of Parallel Computing (ParCo)*, pages 573–580, 2005.
22. Peng Wu, Alexandre E. Eichenberer, Amy Wang, and Peng Zhao. An integrated simdization framework using virtual vectors. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS)*, pages 169–178, 2005.
23. Hans P. Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.