Automatic SIMD Vectorization of Chains of Recurrences

Yixin Shou and Robert A. van Engelen*

Department of Computer Science
and School of Computational Science
Florida State University
Tallahassee, FL 32306
{shou,engelen}@cs.fsu.edu

ABSTRACT

Many computational tasks require repeated evaluation of functions over structured grids, such as plotting in a coordinate system, rendering of parametric objects in 2D and 3D, numerical grid generation, and signal processing. In this paper, we present a method and toolset to speed up closed-form function evaluations over grids by vectorizing Chains of Recurrences (CR). CR forms of closed-form functions require fewer operations to evaluate per grid point. However, the present CR formalism makes CR forms inherently non-vectorizable due to the dependences carried from one point to the next. To address this limitation, we developed a new decoupling method for the CR algebra to translate math functions into Vector Chains of Recurrences (VCR) forms. The VCR coefficients are packed in short vector registers for efficient execution. Performance results of benchmark functions evaluated in single and double precision VCR forms are compared to the Intel compiler's auto-vectorized code and the high-performance small vector math library (SVML). The results show a significant performance increase of our VCR method over SVML and scalar CRs, from doubling the execution speed to running an order of magnitude faster. An auto-tuning tool for VCR is developed for optimal performance and accuracy.

Categories and Subject Descriptors

G.4 [Mathematical Software]: Parallel and vector implementations—*Optimization*

General Terms

Algorithms, Performance

Keywords

Chains of recurrences, vectorization, short vector SIMD, ILP

*Supported in part by NSF grant CCF-0702435.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.
Copyright 2008 ACM 978-1-60558-158-3/08/06 ...\$5.00.

1. INTRODUCTION

Many computational tasks in numerical, visualization, and engineering applications require the repeated evaluation of functions over structured grids, such as plotting in a coordinate system, rendering of parametric objects in 2D and 3D, numerical grid generation, and signal processing. Thread-level parallelism is typically exploited to speed up the evaluation of closed-form functions across points in a grid.

Another effective optimization that yields good speedups when applicable to structured grids or meshes is short-vector SIMD execution of arithmetic operations, e.g. “array arithmetic”. Virtually all modern general-purpose processor architectures feature instruction sets and instruction set extensions that support short-vector SIMD floating point and integer operations, such as MMX/SSE, 3DNow!, AltiVec, and Cell BE SPU SIMD instructions. Coding with these instruction sets is simplified by the use of software, such as state-of-the-art compilers that automatically SIMD-vectorize computationally intensive loops by restructuring these loops for SIMD vectorization [5] often in combination with highly-optimized vector math library kernels [8].

At the hardware level, advances in algorithms for floating point arithmetic have led to significantly faster execution of math library functions in hardware for both scalar and SIMD instructions, such as trigonometric, square root, logarithmic, and exponential functions. For example, the Intel numerics family (8087, 80287, and 80387) use the fast CORDIC (COordinate, Rotation DIgital Computer) methods [15, 16].

However, repeated execution of numerics instructions for each grid point over a structured grid is still costly, especially in loops over the grid points where the execution latency of the iterated floating point instructions cannot be hidden in the instruction pipeline. On an Intel Pentium 4 processor for example, the execution latency of a numerics instruction typically approaches 200 cycles [9].

An effective technique to decrease operation counts and reduce instruction latencies of repeated function evaluations over regular grids is provided by the Chains of Recurrences (CR) formalism [7, 10, 17], which uses an algebraic approach to translate closed-form functions and math kernels into recurrence forms that in a way implements aggressive loop strength reduction [3]. Any floating point expression composed of math library functions and standard arithmetic can be transformed into a CR form and then optimized to achieve efficient execution. For example, if two functions $f(x)$ and $g(x)$ have CR forms, then $f(x) \pm g(x)$, $f(x) * g(x)$, and $f(g(i))$ have CR forms (with some obvious exceptions).

CR forms of closed-form functions require fewer operations per grid point by reusing the value computed at a previous point to determine the function value at the next point. Unfortunately however, this aspect of the CR formalism prohibits loop parallelization and vectorization that require independent operations across the iteration space.

In this paper, we present a vectorization method based on an algebraic transformation to speed up closed-form function evaluations over structured grids using vector representations of CR forms. To this end, we developed a decoupling method for the CR algebra to translate math functions into Vector Chains of Recurrences (VCR) forms. The VCR coefficients are packed into short vector registers for efficient execution. This approach effectively reduces the instruction counts for short-vector SIMD execution of functions over grids. As a result, our method outperforms separately optimized scalar CR forms and SIMD vectorized closed-form function evaluation.

We validated the performance gains using several benchmark functions evaluated in single and double precision and compared the results to the Intel compiler's auto-vectorized code with the high-performance small vector math library (SVML). The results show a dramatic performance increase of our VCR method over SIMD/SVML auto-vectorization and scalar CRs, ranging from doubling the execution speed to running an order of magnitude faster. In addition, the results also show that the VCR code can significantly increase the level of ILP in modulo scheduling [4] resulting in faster kernels for non-SIMD superscalar processors.

CR optimizations introduce potential roundoff errors that are propagated across the iteration space [19]. Error analysis is required to ensure roundoff errors introduced in CR-based evaluation methods are bounded. To address error propagation, we combined our VCR method with an auto-tuning approach. The VCR code is run to determine error properties and select an optimal vector length for performance. This approach ensures optimal performance with high floating point accuracy.

The remainder of this paper is organized as follows. Section 2 introduces the VCR notation, semantics, and algebraic construction. An auto-tuning approach for fast and safe floating point evaluation is described and implementation details are given. Performance results showing improved SIMD execution and ILP scheduling are presented in Section 3, followed by a discussion of alternative approaches and related work in Section 4. Finally, Section 5 summarizes our conclusions.

2. VECTOR CHAINS OF RECURRENCES

This section introduces the VCR formalism, which is a generalization of the scalar CR formalism [7, 10, 17]. The notation and semantics are presented and an algorithm to symbolically construct VCR forms of math functions is given.

2.1 VCR Notation and Semantics

The key idea of our VCR method is to translate a math function or floating point expression into a set of d decoupled CR forms, where each CR represents the value progression of the math function at an offset $0, \dots, d-1$ and stride $d \geq 1$. The *decoupling factor* d increases the level of parallelism by a factor of d and also slows the propagation of roundoff errors by a factor d .

Definition 1. A d -dimensional VCR form Φ_i^d of a function $f(i)$ evaluated over $i = 0, \dots, n-1$ is denoted by

$$\Phi_i^d = \{\vec{\varphi}_0, \odot_1, \vec{\varphi}_1, \odot_2, \dots, \odot_k, \vec{\varphi}_k\}_i,$$

where $\odot$ are “+” or “*” and $\vec{\varphi}$ are vector coefficients

$$\vec{\varphi}_0 = \begin{pmatrix} \vec{\varphi}_0[0] \\ \vec{\varphi}_0[1] \\ \vdots \\ \vec{\varphi}_0[d-1] \end{pmatrix}; \vec{\varphi}_1 = \begin{pmatrix} \vec{\varphi}_1[0] \\ \vec{\varphi}_1[1] \\ \vdots \\ \vec{\varphi}_1[d-1] \end{pmatrix}; \dots \vec{\varphi}_k = \begin{pmatrix} \vec{\varphi}_k[0] \\ \vec{\varphi}_k[1] \\ \vdots \\ \vec{\varphi}_k[d-1] \end{pmatrix}.$$

Hence, a VCR Φ_i^d has $(k+1) \times d$ scalar values and k operations ($\odot = +$ or $\odot = *$) on d -vectors.

Definition 2. The evaluation of a VCR form Φ_i^d of a function $f(i)$ over $i = 0, \dots, n-1$ is defined by the loop

```

vector[d]  $\vec{v}_0$  =  $\vec{\varphi}_0$ 
vector[d]  $\vec{v}_1$  =  $\vec{\varphi}_1$ 
:
:
vector[d]  $\vec{v}_{k-1}$  =  $\vec{\varphi}_{k-1}$ 
for  $i = 0$  to  $n-1$  step  $d$  do
   $y[i : i+d-1] = \vec{v}_0$ 
   $\vec{v}_0 = \vec{v}_0 \odot_1 \vec{v}_1$ 
   $\vec{v}_1 = \vec{v}_1 \odot_2 \vec{v}_2$ 
  :
  :
   $\vec{v}_{k-1} = \vec{v}_{k-1} \odot_k \vec{\varphi}_k$ 
od

```

Note that the loop computes $y[i] = f(i)$ for $i = 0, \dots, n-1$ in vectors of d values per iteration (assuming d divides n).

A method to construct the VCR form of a function is given in the next section. Here, we give a simple toy example to illustrate VCR evaluation. For $d = 2$, the VCR form of $f(i) = i!$ is

$$\begin{aligned} \Phi_i^2 &= \{\vec{\varphi}_0, *, \vec{\varphi}_1, +, \vec{\varphi}_2, +, \vec{\varphi}_3\}_i \\ &= \left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, *, \begin{pmatrix} 2 \\ 6 \end{pmatrix}, +, \begin{pmatrix} 10 \\ 14 \end{pmatrix}, +, \begin{pmatrix} 8 \\ 8 \end{pmatrix} \right\}_i. \end{aligned}$$

Using Definition 2 we obtain the sequence:

$$\begin{array}{c|cccc} i = & 0 & 2 & 4 & 6 & \dots \\ \hline \begin{pmatrix} y[i] \\ y[i+1] \end{pmatrix} = & \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 2 \\ 6 \end{pmatrix} & \begin{pmatrix} 24 \\ 120 \end{pmatrix} & \begin{pmatrix} 720 \\ 5040 \end{pmatrix} & \dots \end{array}$$

where pairs of values are computed in each iteration. In general, for any $d \geq 2$ a speedup is obtained with a vector-based VCR over scalar CR forms for functions evaluated over structured grids and meshes.

2.2 VCR Construction

We present an algorithm to construct a VCR form of a function for any $d \geq 1$. The case $d = 1$ degenerates the VCR to a scalar CR form published in [7, 10, 17].

Algorithm 1. Given a function $f(i)$ represented as a closed-form expression in symbolic form, where $f(i)$ is to be evaluated over n grid points $i = 0, \dots, n-1$, the VCR form Φ_i^d for $d \geq 1$ is constructed in two steps:

Step 1 Construct the symbolic parametric scalar CR form $\Phi_{i'}(j)$ of $f(di' + j)$ by substituting index i in $f(i)$ by $\{j, +, d\}_{i'}$ followed by the application of the CR algebra to simplify this term:

$$f(i) = f(\{j, +, d\}_{i'}) \xrightarrow{CR} \Phi_{i'}(j),$$

which gives the parametric scalar CR form

$$\Phi_{i'}(j) = \{\varphi_0(j), \odot_1, \varphi_1(j), \odot_2, \dots, \odot_k, \varphi_k(j)\}_{i'}$$

Step 2 Construct the symbolic parametric VCR form

$$\Phi_i^d(j) = \{\bar{\varphi}_0(j), \odot_1, \bar{\varphi}_1(j), \odot_2, \dots, \odot_k, \bar{\varphi}_k(j)\}_i$$

with coefficients defined by

$$\bar{\varphi}_0(j) = \begin{pmatrix} \varphi_0(j) \\ \varphi_0(j+1) \\ \vdots \\ \varphi_0(j+d-1) \end{pmatrix}; \dots \bar{\varphi}_k(j) = \begin{pmatrix} \varphi_k(j) \\ \varphi_k(j+1) \\ \vdots \\ \varphi_k(j+d-1) \end{pmatrix}$$

where the coefficients $\varphi(j)$ were constructed in Step 1.

We prove the correctness of the algorithm.

Theorem 1. Let $\Phi_i^d(j)$, $d \geq 1$, be the VCR form constructed by Algorithm 1 for a given function $f(i)$ defined over n points $i = 0, \dots, n-1$. Then, the loop template defined in Definition 2 for $\Phi_i^d(j)$ with $j = 0$ computes $y[i] = f(i)$ exactly for all $i = 0, \dots, n-1$, assuming exact arithmetic, i.e. in the absence of floating point roundoff.

Proof. From Step 1 we have $f(i) = f(di' + j)$ with $i' = \lfloor \frac{i}{d} \rfloor$ and $j = i \bmod d$ for $i = 0, \dots, n-1$. Let $\Phi_{i'}(j)$ denote the scalar CR form of $f(di' + j)$. Then for each $j = 0, \dots, d-1$ the CR forms $\Phi_{i'}(j)$ of $f(di' + j)$ are evaluated for $i' = 0, \dots, \frac{n}{d} - 1$ to compute $f(i) = f(di' + j) = y[di' + j]$ with the CR loop template [10] (scalar form of Definition 2):

```

for  $j = 0$  to  $d - 1$  do
   $x_0(j) = \varphi_0(j)$ 
   $\vdots$ 
   $x_{k-1}(j) = \varphi_{k-1}(j)$ 
  for  $i' = 0$  to  $\frac{n}{d} - 1$  do
     $y[d*i' + j] = x_0(j)$ 
     $x_0(j) = x_0(j) \odot_1 x_1(j)$ 
     $\vdots$ 
     $x_{k-1}(j) = x_{k-1}(j) \odot_k \varphi_k(j)$ 
  od
od

```

Because the j -loop iterations are independent, we can reorder the loops and use array assignment notation to obtain

```

 $x_0(0 : d-1) = \varphi_0(0 : d-1)$ 
 $\vdots$ 
 $x_{k-1}(0 : d-1) = \varphi_{k-1}(0 : d-1)$ 
for  $i' = 0$  to  $\frac{n}{d} - 1$  do
   $y[d*i' : d*i' + d-1] = x_0(0 : d-1)$ 
   $x_0(0 : d-1) = x_0(0 : d-1) \odot_1 x_1(0 : d-1)$ 
   $\vdots$ 
   $x_{k-1}(0 : d-1) = x_{k-1}(0 : d-1) \odot_k \varphi_k(0 : d-1)$ 
od

```

Taking $i = d*i'$, this loop is equivalent to the loop in Definition 2 for Φ_i^d of $f(i)$. $\square$

In Step 1 of the algorithm, the CR algebra described in [7, 10, 17] is used to simplify the parametric scalar CR form of f . A comprehensive list of CR algebra rules can be found in [13]. In certain cases, the exhaustive application of CR rules does not simplify to a single CR form but results in CR-expressions [7] which contain multiple CR forms (see Example 3 at the end of this section). These CR forms can be separately evaluated in loops or in fused loops.

Parameter j in the symbolic form $\Phi_i^d(j)$ is also used to generate blocked versions of the VCR loops (see Section 2.3).

VCR construction is applicable to factorials, sums, products, polynomials, exponentials, transcendental, and compositions. We conclude this section with some examples.

Example 1. (From the example shown in Section 2.1) Consider $f(i) = i!$ for $i = 0, \dots, n-1$. Let $d = 2$. Then, $f(i) = f(2i' + j) = f(\{j, +, 2\}_{i'}) = \{j, +, 2\}_{i'}!$. Applying the CR algebra $\{j, +, 2\}_{i'}! \xrightarrow{CR} \{j!, *, j^2 + 3j + 2, +, 4j + 10, +, 8\}_{i'}$. The VCR Φ_i^2 is constructed from the coefficients $\varphi_0(j) = j!$, $\varphi_1(j) = j^2 + 3j + 2$, $\varphi_2(j) = 4j + 10$, and $\varphi_3(j) = 8$, giving

$$\bar{\varphi}_0 = \begin{pmatrix} j! \\ (j+1)! \end{pmatrix}; \bar{\varphi}_1 = \begin{pmatrix} j^2 + 3j + 2 \\ j^2 + 5j + 6 \end{pmatrix}; \bar{\varphi}_2 = \begin{pmatrix} 4j + 10 \\ 4j + 14 \end{pmatrix}; \bar{\varphi}_3 = \begin{pmatrix} 8 \\ 8 \end{pmatrix}$$

and we set $j = 0$ (non-blocked loops), which simplifies to

$$\Phi_i^2 = \left\{ \begin{pmatrix} 1 \\ 1 \end{pmatrix}, *, \begin{pmatrix} 2 \\ 6 \end{pmatrix}, +, \begin{pmatrix} 10 \\ 14 \end{pmatrix}, +, \begin{pmatrix} 8 \\ 8 \end{pmatrix} \right\}_i$$

The value sequence of Φ_i^2 is shown in Section 2.1.

Example 2. Consider $f(i) = ri^2$ for $i = 0, \dots, n-1$. Let $d = 4$. Then, $f(i) = f(4i' + j) = f(\{j, +, 4\}_{i'}) = r\{j, +, 4\}_{i'}^2$

$$\begin{aligned} r\{j, +, 4\}_{i'}^2 &\xrightarrow{CR} r\{j, +, 4\}_{i'}\{j, +, 4\}_{i'} \\ &\xrightarrow{CR} r\{j^2, +, 8\{j, +, 4\}_{i'} + 16\}_{i'} \\ &\xrightarrow{CR} \{rj^2, +, 8rj + 16r, +, 32r\}_{i'} \end{aligned}$$

The VCR Φ_i^4 is constructed from the symbolic coefficients $\varphi_0(j) = rj^2$, $\varphi_1(j) = 8rj + 16r$, and $\varphi_2(j) = 32r$, giving

$$\bar{\varphi}_0(j) = r \begin{pmatrix} j^2 \\ j^2 + 2j + 1 \\ j^2 + 4j + 4 \\ j^2 + 6j + 9 \end{pmatrix}; \bar{\varphi}_1(j) = r \begin{pmatrix} 8j + 16 \\ 8j + 24 \\ 8j + 32 \\ 8j + 40 \end{pmatrix}; \bar{\varphi}_2(j) = r \begin{pmatrix} 32 \\ 32 \\ 32 \\ 32 \end{pmatrix}$$

and we set $j = 0$ to obtain the Φ_i^4 coefficients

$$\Phi_i^4 = \left\{ \begin{pmatrix} 0 \\ r \\ 4r \\ 9r \end{pmatrix}, +, \begin{pmatrix} 16r \\ 24r \\ 32r \\ 40r \end{pmatrix}, +, \begin{pmatrix} 32r \\ 32r \\ 32r \\ 32r \end{pmatrix} \right\}_i$$

By Definition 2 we obtain the sequence

$i =$	0	4	8	12	...
$\begin{pmatrix} y[i] \\ y[i+1] \\ y[i+2] \\ y[i+3] \end{pmatrix}$	$\begin{pmatrix} 0 \\ r \\ 4r \\ 9r \end{pmatrix}$	$\begin{pmatrix} 16r \\ 25r \\ 36r \\ 49r \end{pmatrix}$	$\begin{pmatrix} 64r \\ 81r \\ 100r \\ 121r \end{pmatrix}$	$\begin{pmatrix} 144r \\ 169r \\ 196r \\ 225r \end{pmatrix}$	...

Example 3. Consider $f(i) = \sin(hi)$ for $i = 0, \dots, n-1$. For $d \geq 1$ $f(i) = f(di' + j) = f(\{j, +, d\}_{i'}) = \sin(h\{j, +, d\}_{i'})$

$$\begin{aligned} &\xrightarrow{CR} \Re(\{\tfrac{1}{2} \sin(hj) - \tfrac{1}{2} \cos(hj)I, *, \cos(dh) + \sin(dh)I\}_{i'}) + \\ &\quad \Re(\{\tfrac{1}{2} \sin(hj) + \tfrac{1}{2} \cos(hj)I, *, \cos(dh) - \sin(dh)I\}_{i'}) \end{aligned}$$

where I is the imaginary unit and $\Re(z)$ is the real part of a complex number z . Note that the CR forms represent exponential sequences in the complex domain. Let $d = 2$, then

$$\Phi_i^2 = \Re(\left\{ \begin{pmatrix} -\frac{1}{2}I \\ \gamma \end{pmatrix}, *, \begin{pmatrix} \alpha + \beta \\ \alpha + \beta \end{pmatrix} \right\}_i) + \Re(\left\{ \begin{pmatrix} \frac{1}{2}I \\ \delta \end{pmatrix}, *, \begin{pmatrix} \alpha - \beta \\ \alpha - \beta \end{pmatrix} \right\}_i)$$

with $\alpha = \cos(2h)$, $\beta = \sin(2h)I$, $\gamma = \frac{1}{2} \sin h - \frac{1}{2} I \cos h$, and $\delta = \frac{1}{2} \sin h + \frac{1}{2} I \cos h$. This VCR requires only one vector addition and two complex multiplications per two grid points.

```

for  $j = 0$  to  $n - (n \bmod (b * d)) - 1$  step  $(b * d)$  do
   $\vec{v}_0 = \vec{\varphi}_0(j)$ 
   $\vec{v}_1 = \vec{\varphi}_1(j)$ 
   $\vdots$ 
   $\vec{v}_{k-1} = \vec{\varphi}_{k-1}(j)$ 
  for  $i = j$  to  $j + (b * d) - 1$  step  $d$  do
     $y[i : i+d-1] = \vec{v}_0$ 
     $\vec{v}_0 = \vec{v}_0 \odot_1 \vec{v}_1$ 
     $\vec{v}_1 = \vec{v}_1 \odot_2 \vec{v}_2$ 
     $\vdots$ 
     $\vec{v}_{k-1} = \vec{v}_{k-1} \odot_k \vec{\varphi}_k(j)$ 
  od
od
 $j = \lfloor n / (b * d) \rfloor * b * d$ 
 $\vec{v}_0 = \vec{\varphi}_0(j)$ 
 $\vec{v}_1 = \vec{\varphi}_1(j)$ 
 $\vdots$ 
 $\vec{v}_{k-1} = \vec{\varphi}_{k-1}(j)$ 
for  $i = j$  to  $n - (n \bmod d) - 1$  step  $d$  do
   $y[i : i+d-1] = \vec{v}_0$ 
   $\vec{v}_0 = \vec{v}_0 \odot_1 \vec{v}_1$ 
   $\vec{v}_1 = \vec{v}_1 \odot_2 \vec{v}_2$ 
   $\vdots$ 
   $\vec{v}_{k-1} = \vec{v}_{k-1} \odot_k \vec{\varphi}_k(j)$ 
od
for  $i = n - (n \bmod d)$  to  $n - 1$  do
   $y[i] = \vec{v}_0[i \bmod d]$ 
od

```

Figure 1: Blocked VCR Loop Template

2.3 VCR Loop Blocking

Zima et al. [20] analyzed the error characteristics of real and complex-valued CR forms by considering two primary CR categories, “pure-sum CR” and “pure-product CR”. The results indicate that the error of pure-sum CR forms (polynomials) increases with increasing recurrence iteration distance, but is constrained and independent of the actual function value at each point. Thus, evaluation of polynomials with CR forms is numerically stable. The error in exponential CR forms is constrained, but depends both on the iteration distance and on the function values. Evaluations of these and mixed CR forms using (complex) floating-point arithmetic may not be numerically stable.

Therefore, blocking of VCR loops is essential. It forces re-initialization of the VCR sequence to stop error propagation. Figure 1 shows our blocked VCR loop template. The block parameter b is used to bound the recurrence iteration length. After b recurrence steps in the inner i -loop, the recurrences are re-initialized in the j -loop. The remainder of the code handles the case when n is not a multiple of $b * d$.

Theoretically, the block size b should be an inverse function of the growth of the relative error. That is, a small growth in relative errors generally yields a larger block size. Because of the unpredictability of the error propagation in non-polynomial CR forms, the block size b must be determined by empirical evaluation.

2.4 VCR Auto-Tuning

We use an auto-tuning approach for the generated code to optimize decoupling factor d for speed and block size b for floating-point accuracy. A set of parametric loops is gen-

Function	$\alpha(n)$	
	Single Precision	Double Precision
<i>Poly3</i>	$3.8 \cdot 10^{-9} n$	$8.2 \cdot 10^{-18} n$
<i>Spline</i>	$5.0 \cdot 10^{-9} n^2$	$6.0 \cdot 10^{-18} n^2$

Table 1: Interpolated Maximum Relative Errors

erated for $d = 1, 2, 4, 8$ to find the best speedup and largest b . Small values of b negatively impact performance, while b cannot be too large to stay within the error threshold.

Increasing d yields better SIMD vector utilization. Increasing d beyond 8 typically results in performance degradation due to increased vector register pressure to hold VCR coefficients in vector registers, possibly leading to register spill. Increasing d also increases the level of ILP in modulo scheduling [4], which may result in faster kernels for non-SIMD superscalar processors. The decreased number of anti/flow dependences of the VCR form updates compared to scalar CR forms allows the scheduler to schedule instructions more optimally. Here we assume the kernel is small, i.e. a loop to execute a few math functions over a grid.

The basic approach to find optimal d and b for ILP is the same as for SIMD auto-tuning. More specifically, given a maximum relative error threshold $\varepsilon_{\max}$, the goal is to find a block size b such that the VCR evaluation error $\varepsilon_b \leq \varepsilon_{\max}$ is bounded. Because performance typically increases with larger b , results are computed faster when the error tolerance is higher. Thus, b should be maximized while $\varepsilon_b \leq \varepsilon_{\max}$.

The auto-tuning phase determines b as follows. A blocked VCR profile loop is generated for each value of $d = 1, 2, 4, 8$ and a starting value of b (in this paper $b = 10^6$ which is sufficiently large) and a given maximum n to compute

$$\varepsilon_b = \max_{i=0, \dots, n-1} \left| \frac{f(i) - \Phi_i^d}{f(i)} \right|.$$

To refine b , this process repeats for exponentially diminishing values of b until $\varepsilon_b \leq \varepsilon_{\max}$. The performance of the best combination of d and b is selected.

For pure-sum VCR forms, which are polynomials and combinations of polynomials such as spline calculations, error analysis is simpler. The error of the scalar CR form (with $d = 1$) is computed for increasing n and tabulated and interpolated to determine the error bound $\alpha(n)$. Then the error for a choice of b and d can be determined from $\varepsilon_b = \alpha(b)$. Because we require $\varepsilon_b \leq \varepsilon_{\max}$, the optimal block size b for a given d can be determined from $b = \alpha^{-1}(\varepsilon_{\max})$.

Consider for example the $\alpha(n)$ values of a third-order polynomial *Poly3* and a bicubic *Spline* function shown in Table 1. Suppose that four digits of precision are required, i.e. $\varepsilon_{\max} \leq 10^{-5}$. Then $b = 2.6 \cdot 10^3$ for *Poly3* and $b = 45$ for *Spline* using single precision floating point, and $b = 1.2 \cdot 10^{12}$ for *Poly3* and $b = 1.3 \cdot 10^6$ for *Spline* using double precision.

2.5 Shuffle-Based SIMD Vectorization

In this paper we compare the shuffle-based CR vectorization method suggested in [20] to our VCR approach. When all k operators in a CR form of length k are identical, i.e. (+) or (*), then this regularity can be exploited with a vector operation of length $k + 1$.

Figure 2 (b) shows the SSE code for a cubic polynomial (CR length $k=3$). A copy of vector $\text{crv}=[\text{cr0}, \text{cr1}, \text{cr2}, \text{cr3}]$ is made to vector tmp using `_mm_sub_ss` such that $\text{tmp}[0]=0$, then tmp is rotated $\text{tmp}=[\text{cr1}, \text{cr2}, \text{cr3}, 0]$ and added to crv .

Figure 2: Shuffle-Based CR Vectorization

```

cr0r = 0.0;   cr0i = -0.5;   cr1r = 0.0;   cr1i = 0.5;
cr2r = cos(h); cr2i = sin(h); cr3r = cr2r; cr3i = -cr2i;
for (i = 0, i < n; i++)
{
  y[i] = cr0r + cr1r;
  tmp1 = cr0r*cr2r - cr0i*cr2i; cr0i = cr0r*cr2i + cr0i*cr2r;
  tmp2 = cr1r*cr3r - cr1i*cr3i; cr1i = cr1r*cr3i + cr1i*cr3r;
  cr0r = tmp1; cr1r = tmp2;
}

```

Figure 3: Scalar CR Code of *Sine* Benchmark

The shuffle-based method is limited to SIMD vectorization of polynomials and exponential CR forms, not mixed forms, e.g. factorials. In addition, the method is only optimal when the length of the CR is a multiple of the SIMD register size. This method is not further considered until Section 3.

2.6 Implementation Details

We implemented the function translation and auto-tuned VCR code generation techniques with the help of the Ctadel system [14] (implemented in SWI-Prolog [1]), which has an algebraic optimizer, term rewriting system, and code generator. The CR rules [13] were implemented along with an SSE intrinsics code generator for C.

The symbolic form of a math expression containing library function calls to be evaluated over variable i is entered, together with a constant decoupling factor $d \geq 1$. We generate two loop codes using Ctadel, one for profile-based auto-tuning that also includes the original closed-form function to compute the maximum relative error, and the optimized VCR blocked loop with SSE intrinsics. The final code is easily integrated into the user's code as a precomputation.

2.7 Example Application

Consider $f(i) = \sin(hi)$ from Example 3 of Section 2.2 evaluated over a grid $i = 0, \dots, n-1$ in single-precision floating point. The scalar CR loop is shown in Figure 3. As discussed, the standard CR loop [7, 17] is not vectorizable due to cross iteration dependences.

For $d = 4$, the VCR form for $\sin(hi)$ is generated that represent four parallel running sequences as illustrated in Figure 4. Figure 5 shows the corresponding 4-wide vectorized VCR code for $\sin(hi)$ with Intel SSE instructions (loop blocking and epilogue code are elided for clarity). The VCR coefficients of the VCR form are stored into the 128-bit vec-

Figure 4: *Sine* Sequences for $d = 4$

```

register __m128 crv0r, crv1r, crv2r, crv3r;
register __m128 crv0i, crv1i, crv2i, crv3i;
register __m128 crv0rt, crv1rt, vt1, vt2, *yv;
...
ch = cos(h);      sh = sin(h);
c2h = 2*ch*cch - 1;  s2h = 2*sh*sch;
c4h = 2*c2h*c2h - 1;  s4h = 2*s2h*s2h;
c4h = c4h;         s4h = s4h;
crv0r = _mm_set_ps(sh*cch*cch+0.5*sh*c2h, sh*cch, 0.5*sh, 0.0);
crv1r = _mm_set_ps(sh*cch*cch+0.5*sh*c2h, sh*cch, 0.5*sh, 0.0);
crv2r = _mm_set_ps(c4h, c4h, c4h, c4h);
crv3r = _mm_set_ps(c4h, c4h, c4h, c4h);
crv0i = _mm_set_ps(-0.5*cch*c2h+sh*cch*sh, -0.5*c2h, -0.5*cch, -0.5);
crv1i = _mm_set_ps(0.5*cch*c2h-sh*cch*sh, 0.5*c2h, 0.5*cch, 0.5);
crv2i = _mm_set_ps(s4h, s4h, s4h, s4h);
crv3i = _mm_set_ps(s4h, s4h, s4h, s4h);
for (i = 0, i < n/4; i++)
{
  yv[i] = _mm_add_ps(crv0r, crv1r);
  vt1 = _mm_mul_ps(crv0r, crv2r); vt2 = _mm_mul_ps(crv0i, crv2i);
  crv0rt = _mm_sub_ps(vt1, vt2);
  vt1 = _mm_mul_ps(crv0r, crv2i); vt2 = _mm_mul_ps(crv0i, crv2r);
  crv0i = _mm_add_ps(vt1, vt2);
  vt1 = _mm_mul_ps(crv1r, crv3r); vt2 = _mm_mul_ps(crv1i, crv3i);
  crv1rt = _mm_sub_ps(vt1, vt2);
  vt1 = _mm_mul_ps(crv1r, crv3i); vt2 = _mm_mul_ps(crv1i, crv3r);
  crv1i = _mm_add_ps(vt1, vt2);
  crv0r = crv0rt; crv1r = crv1rt;
}

```

Figure 5: Generated *Sine* Code for $d = 4$ (Unblocked)

tor registers. As expected, four values are computed and stored in array yv per iteration.

As can be seen in the code in Figure 5, additional optimizations have been applied to reduce the evaluation of trigonometric functions. Simplification rules are integrated in Ctadel to simplify expressions and remove redundant computations. For example, the following four formulas for sine and cosine functions are adopted in Ctadel:

$$\begin{aligned}
\cos(-h) &= \cos(h) \\
\sin(-h) &= -\sin(h) \\
\cos(2 * h) &= 2 * \cos(h) * \cos(h) - 1 \\
\sin(2 * h) &= 2 * \sin(h) * \cos(h)
\end{aligned}$$

Figure 6 illustrates the increased levels of parallelism with $d = 8$ to evaluate $\sin(hi)$ over $i = 0, \dots, n-1$. This 8-way, 4-wide VCR code uses twice as many vector registers compared to the implementation of 4-way, 4-wide vectorization. While the performance of the code is typically better due to improved scheduling with higher ILP, the extra register pressure could lead to spill which degrades performance.

The performance results and floating point accuracy of this example application and other benchmark math functions will be presented in the next section and compared to the performance of scalar CR forms, shuffle-based vector CR forms, and the Intel compiler's advanced auto-vectorization.

Figure 6: *Sine* Sequences for $d = 8$

Benchmark	Description
<i>Poly3</i>	Cubic polynomial
<i>Spline</i>	Bicubic spline
<i>Bin15</i>	Binomial coefficient $\binom{i}{15}$
<i>Sine</i>	Sine function $\sin(hi)$
<i>Sinh</i>	Hyperbolic sine $\sinh(hi)$
<i>Exp</i>	Exponential $\exp(hi(i+1)/2)$

Table 2: Functions used in Benchmark Codes

Class	Description
<i>Scalar</i>	Original closed-form w/o vectorization
<i>SVML</i>	Original closed-form vectorized w/ SVML
<i>Scalar CR</i>	Serial scalar CR form of a function
<i>VCR4.4</i>	4-way 4-wide VCR vectorization
<i>VCR8.4</i>	8-way 4-wide VCR vectorization
<i>VCR2.2</i>	2-way 2-wide VCR vectorization
<i>VCR4.2</i>	4-way 2-wide VCR vectorization
<i>VCR4.1</i>	Serial CR decoupled 4 times (=4 CR forms)
<i>sCR1.4</i>	1-way 4-wide shuffle-based CR vectorization
<i>sCR4.4</i>	4-way 4-wide shuffle-based CR vectorization

Table 3: Benchmark Optimization Classification

3. RESULTS

In this section we present the performance results of our VCR technique applied to benchmark functions that represent common classes of math operations.

3.1 Benchmark Classification

The benchmark functions listed in Table 2 are used in the performance experiments. These benchmark functions were selected as representable examples of classes of common math operations found in numerical applications. There are two main classes: *polynomial VCR forms* (*Poly3*, *Spline*, and *Bin15* which is a 15th order polynomial) and *exponential VCR forms* (*Sine*, *Sinh*, and *Exp*). The functions are evaluated over the interval $i = 0, \dots, n-1$ for $n \in [1, \dots, 10^6]$.

Table 3 classifies the benchmark codes according to the type of optimization applied in the performance comparisons. The “*Scalar*” codes run the original closed-form benchmark function. The “*SVML*” code represents the original closed-form function auto-vectorized by the Intel compiler (option `-axW`) with SVML. The “*Scalar CR*” codes refer to the use of standard scalar CR forms published in [7, 20] to optimize function evaluations. The “*VCRd.w*” codes are produced using the techniques described in this paper, where d is the decoupling factor and w is the vector register length (d must be a multiple of w for VCR). The “*sCRd.w*” codes denote shuffle-based CR vectorized forms for performance comparison, as described in Section 3.4.

Because floating-point operation counts (flops) vary significantly between the CR and non-CR optimizations, the performance is expressed in *normalized GFLOPS*. In the sequel, GFLOPS is the *total standardized flops* over the *total execution time* t in seconds: $\text{GFLOPS} = \frac{\text{flops}}{t} \cdot 10^{-9}$. The basis of the standardized flops is the *Scalar CR* code, which uses the least number of flops to compute the function per grid point. For example, the total standardized flops = $3n$ for *Poly3* over n points (see Figure 2(a)), regardless of the vectorization and/or CR optimizations. This ensures a fair comparison of the optimizations by expressing performance as the inverse of execution time scaled by problem size.

Benchmark	Maximum Relative Error	
	Single-precision	Double-precision
<i>Poly3</i>	$3.72 \cdot 10^{-07}$	$8.15 \cdot 10^{-16}$
<i>Spline</i>	$1.62 \cdot 10^{-05}$	$4.90 \cdot 10^{-14}$
<i>Bin15</i>	$2.96 \cdot 10^{-07}$	$2.07 \cdot 10^{-16}$
<i>Sine</i>	$1.37 \cdot 10^{-04}$	$3.62 \cdot 10^{-13}$
<i>Sinh</i>	$5.26 \cdot 10^{-06}$	$8.81 \cdot 10^{-15}$
<i>Exp</i>	$1.76 \cdot 10^{-06}$	$3.00 \cdot 10^{-14}$

Table 4: Maximum Relative Error for $n = 100$

3.2 Floating Point Accuracy

The maximum relative error of the scalar CR form (with $d = 1$) for $n = 100$ steps is given for single-precision and double-precision floating point for each benchmark in Table 4. The table shows that the relative error of CR evaluation for these benchmarks is quite small, especially for double precision. In general, using double precision to compute single precision results with the VCR method for large n appears to be desirable, though this effectively halves the vector register width and thus reduces performance. A VCR blocking size b is selected based on the error properties of a specific function as described in Sections 2.3 and 2.4.

Larger decoupling factors d limit the error propagation. For example, suppose $d = 8$ and $b = 100$ recurrence steps are taken. Then 800 grid points are efficiently computed with an accuracy that is listed in Table 4. Results discussed in the next section indicate that the performance of our VCR method exceeds other methods for as few as 10 grid points and reaches a maximum for about 1,000 grid points. Thus, $b = 100$ is a realistic bound to keep the error low and performance high with $d = 4$ or $d = 8$.

3.3 SIMD Performance Results

The performance results presented in this section were obtained using the Intel C++ compiler v9.1 with options “`icc -O3 -restrict -fno-inline -axW`”. The compiler uses a clever technique to vectorize math functions using SSE2 instructions and SVML by expanding the series computations [8]. Experiments were performed on a Intel Dual Core Xeon 5160 3GHz running Linux Fedora Core 7. Each benchmark was run repeatedly and the average performance recorded. The execution time variance was low $\approx 1\%$.

Figure 7 compares the performance in GFLOPS for the benchmarks listed in Table 2, for $n = 1,000$ grid points and $n = 100,000$ grid points. Recall that the accuracy of CR methods is high for $n = 1,000$, but can be poor for $n = 100,000$, necessitating blocking, e.g. in intervals of 1,000 points resulting in the performance shown for $n = 1,000$.

A dramatic increase in performance is observed for the *SVML* code compared to the *Scalar* code, due to the efficient auto-vectorization of the scalar code by the Intel compiler. However, the Intel compiler failed to vectorize *Bin15*.

The use of CR-based methods to reduce operation counts also has a dramatic impact on performance. The *Scalar CR* codes are significantly faster than the *Scalar* codes of the original benchmark functions (both codes are highly optimized, e.g. with loop unrolling). However, operation reduction alone is not sufficient to achieve high performance as can be seen in the figure by comparing *SVML* to *Scalar CR*. In fact, the speed of the *SVML* code for *Poly3*, *Spline*, and *Sine* is generally faster than the *Scalar CR* code.

Figure 7: Performance Comparison of VCR for $n = 1,000$ and $n = 100,000$ (Intel Dual Core Xeon 5160 3GHz)

Figure 8: Sine Single and Double Precision Performance Comparison (Intel Dual Core Xeon 5160 3GHz)

The vectorization of CR forms using our VCR method gives the highest overall performance improvement as can be seen in Figure 7. The *VCR4.4* code executes at least twice as fast as the *SVML* and *Scalar CR* codes, peaking at an almost fourfold speedup for *Bin15* and *Sine*. Further performance improvements are obtained with *VCR8.4*. For $n = 1,000$, the exponential VCR benchmarks *Sine*, *Sinh*, and *Exp* are faster. For $n = 100,000$ the performance of *VCR8.4* is faster overall. The level of parallelism is increased by a factor $p = d/w = 2$ for *VCR8.4*. Therefore, the instruction scheduler has fewer dependences to take into account, which leads to a higher instructions per cycle (IPC) ratio.

To investigate the scalability of the performance over the interval $[10, \dots, 10^6]$ for *SVML*, *Scalar CR*, and *VCR*, we plotted the normalized MFLOPS of the *Sine* benchmark in Figure 8 for single and double-precision floating point. For $n \geq 50$, the *VCR8.4* and *VCR4.2* codes are the fastest, for the single and double-precision experiments, respectively. The lower performance for smaller n is caused by the initialization overhead to setup the VCR vectors, where the initialization cost increases with larger d . Note that for the single-precision case the *VCR4.4* code is about a factor 3 to 4 faster than the *Scalar CR* code, showing close-to-perfect vec-

torization cost reduction and scalability for $n \geq 500$. Similarly, for the double-precision case the *VCR2.2* code is about a factor 2 faster than the *Scalar CR* code for $n \geq 500$.

The performance of the *VCR* codes drop after $n = 500,000$ due to cache capacity effects, while the *SVML* code performance drops much earlier due to temporary vector memory storage and cache effects. The apparent jitter observed in the graphs for the *VCR4.4* and *VCR8.4* single-precision code is consistent and appears to be related to the use of more aggressive optimization options with the Intel compiler, indicating that certain points benefit from increased performance due to loop optimizations such as unrolling.

The *VCR8.4* code exhibits higher levels of independence between iterations than *VCR4.4* (and similar for *VCR4.2* versus *VCR2.2*). This significantly increases the computational throughput for the *Sine* benchmark for the single precision case and more dramatically for the double precision case as observed in Figure 8.

Note that the performance of double-precision *SVML* code has decreased to around half of the performance of the single precision *SVML* code, which is due to the fact that effective SIMD vector length is reduced by half. Similarly, the performance of the *VCR2.2* code and the *VCR4.2* code also

```

register __m128 cr0, cr1, cr2, cr3, tmp;
int s = _MM_SHUFFLE(0, 3, 2, 1);
cr0 = ...; cr1 = ...; cr2 = ...; cr3 = ...;
for (i = 0; i < n; i++)
{
    _mm_store_ss(&x[i], cr0);
    tmp = _mm_move_ss(cr0, cr1); tmp = _mm_shuffle_ps(tmp, tmp, s);
    cr0 = _mm_add_ps(cr0, tmp);
    tmp = _mm_move_ss(cr1, cr2); tmp = _mm_shuffle_ps(tmp, tmp, s);
    cr1 = _mm_add_ps(cr1, tmp);
    tmp = _mm_move_ss(cr2, cr3); tmp = _mm_shuffle_ps(tmp, tmp, s);
    cr2 = _mm_add_ps(cr2, tmp);
    tmp = _mm_sub_ss(cr3, cr3); tmp = _mm_shuffle_ps(tmp, tmp, s);
    cr3 = _mm_add_ps(cr3, tmp);
}

```

```

register __m128 crv0, crv1, crv2, crv3, tmp;
int s = _MM_SHUFFLE(0, 3, 2, 1);
crv0 = ...; crv1 = ...; crv2 = ...; crv3 = ...;
for (i = 0; i < n; i += 4)
{
    _mm_store_ss(&y[i], crv0); _mm_store_ss(&y[i+1], crv1);
    _mm_store_ss(&y[i+2], crv2); _mm_store_ss(&y[i+3], crv3);
    tmp = _mm_sub_ss(crv0, crv0); tmp = _mm_shuffle_ps(tmp, tmp, s);
    crv0 = _mm_add_ps(crv0, tmp);
    tmp = _mm_sub_ss(crv1, crv1); tmp = _mm_shuffle_ps(tmp, tmp, s);
    crv1 = _mm_add_ps(crv1, tmp);
    tmp = _mm_sub_ss(crv2, crv2); tmp = _mm_shuffle_ps(tmp, tmp, s);
    crv2 = _mm_add_ps(crv2, tmp);
    tmp = _mm_sub_ss(crv3, crv3); tmp = _mm_shuffle_ps(tmp, tmp, s);
    crv3 = _mm_add_ps(crv3, tmp);
}

```

Figure 9: *sCR4.1* Code of the *Bin15* Benchmark and *sCR4.4* Code of the *Poly3* Benchmark

Figure 10: Performance Comparison of Shuffle-Based CR Vectorization for the Polynomial Benchmarks with $n = 1,000$ and for *Poly3* with $n = 10, \dots, 10^6$ (Intel Dual Core Xeon 5160 3GHz)

decreased to half of the performance of the *VCR4.4* code and the *VCR8.4* code, respectively. The performance of the serial version *Scalar CR* is about the same as its single precision version, which is not surprising given that the code is not vectorized.

Overall, the performance of the VCR code is superior to any of the other optimization methods, sometimes even an order of magnitude faster than the original scalar code and *SVML*-vectorized code. The VCR transformation effectively combines CR-based operation reduction with SIMD vectorization. The choice of decoupling factor d that gives the best speedup of VCR depends on the function to optimize and is difficult to predict which leads us to the auto-tuning approach. Note that d should be a multiple of the SIMD vector register length w and reasonable values are therefore $d = 2$, $d = 4$, and $d = 8$.

3.4 Shuffle-Based CR Performance Results

In this section, the performance results of the shuffle-based CR vectorization, denoted *sCR*, is compared. Experiments were performed on a Intel Dual Core Xeon 5160 3GHz running Linux Fedora Core 7, using the Intel C++ compiler with compiler flags “icc -O3 -restrict -fno-inline -axW”. Recall that the shuffle-based CR vectorization approach is only applicable to polynomial benchmarks *Poly3*, *Spline*, and *Bin15*.

The optimized design of the SSE code for shuffle-based CR operations illustrates the use of advanced instructions and some tricks to reduce the operation count to the bare minimum. The transformation from scalar CR code to shuffle-based *sCR* code was shown in Figure 2. A more elaborate example of *sCR1.4* code for *Bin15* is shown in Figure 9, which further draws on specialized SSE instructions to manipulate CR coefficients packed in vector registers.

To further speed up *sCR* code, we used a decoupling factor to generate independent execution sequences that increases the level of ILP. The SSE code for the *sCR* optimization with a decoupling factor $d = 4$, the *sCR4.4* shuffle-based CR vectorization of benchmark *Poly3* is shown in Figure 9. The code essentially replicates the shuffle-based template of Figure 2 by applying our decoupling technique to the shuffle-based CR method.

Figure 10 summarizes the performance for all three polynomial benchmarks for $n = 1,000$ grid points. Note that the Intel compiler failed to vectorize *Bin15*. Except for *Bin15*, the Intel *SVML*-vectorized code is fastest. The *Poly3 sCR1.4* code has the worst performance among these versions. The shuffle operation used in this version is the reason of the slow-down. Shuffle operations that involve a data reorganization in vector registers are usually expensive and appear to be more costly than the non-vector floating point operations on floating point registers in the *Scalar CR*

Figure 11: Performance Comparison of VCR-Enhanced ILP (UltraSPARC IIIi 1.2GHz)

code. Increasing the ILP appears to help for long CR forms, e.g. the 15th order polynomial *Bin15* code.

To investigate the scalability of the performance of *sCR* over the interval $[10, \dots, 10^6]$, the performance results for the *Poly3* benchmark are also shown in Figure 10. As can be expected, the *Poly3 sCR4.4* code shows an almost four-fold performance increase over the *Poly3 sCR1.4* code due to higher ILP.

The overall performance of *sCR* is disappointing compared to *SVML*. Thus, *sCR* is also much slower than our proposed VCR method (since VCR is overall faster than *SVML*). Given the low performance of *sCR* and its limited applicability, The *sCR* optimization is not a viable alternative. However, new SSE instructions such as a “shift-add” that would fit the CR update operations could make the performance of *sCR* more competitive.

3.5 Superscalar ILP Enhancement Results

The performance results presented in this section were obtained using the Sun Studio 12 C compiler with compiler options “`suncc -fast -fma=fused -xrestrict -xalias_level=layout -xinline=no -xarch=native`”. Note that the “`-fma=fused`” option is used to improve the performance of the code with fused multiply-add (FMA) instructions, when possible. Experiments were performed on an UltraSPARC IIIi 1.2GHz running Sun Solaris 10. The UltraSPARC IIIi processor is a high performance, highly-integrated 4-issue superscalar processor implementation of the 64-bit SPARC-V9 RISC architecture, supporting a 64-bit virtual address space. We did not use the UltraSPARC VIS instruction set for SIMD short vector operations, which supports only integer and fixed-point operand types. Therefore, any improvement is a result of the improved utilization of ILP in modulo scheduling.

The 4-issue superscalar processor performance benefits from increased levels of ILP. Thus modulo scheduling is an essential compiler optimization for small loop kernels. The *Scalar CR* code exhibits loop-independent anti dependences between the CR updates and cross iteration dependences (see Figure 2(a) for example) that limit the effectiveness of the modulo scheduler to schedule instructions in parallel. The best performance is obtained with the lowest steady-

Measurement	<i>Poly3</i> Benchmark		
	<i>Scalar</i>	<i>Scalar CR</i>	<i>VCR4.1</i>
<i>Unroll factor</i>	8	8	2
<i>Steady-state cycle count</i>	5	4	3 (12 per 4 pts)
<i>Floating point operations per iteration</i>	3 FMA + 2 FPadds	3 FPadds	12 FPadds (per 4 points)

Table 5: Static Code Statistics of Three *Poly3* Codes

state cycle count (also referred to as the initiation interval of the kernel), assuming no stalls.

Figure 11 summarizes the performance of all benchmark codes, for $n = 1,000$ grid points and $n = 100,000$ grid points. The performance of the transcendental math library functions on this machine is very poor (the *Scalar* codes), making the data for *Sinh* and *Exp* barely visible in the graph. The difference is marginally better for higher $n = 100,000$, because of the relatively lower overhead of CR initialization cost and relatively lower prologue and epilogue costs in the modulo schedule.

The hypothesized increased ILP of our VCR method is indeed verified by the results in Figure 11. The *VCR 4.1* code (decoupling factor $d = 4$ and vector width $w = 1$) achieves best performance for all benchmark functions except *Bin15*. The modulo scheduler of the Sun Studio compiler failed to construct a schedule for the *Bin15* scalar code, due to the high floating-point register pressure brought by four decoupled 15-order polynomial CR functions requiring the use of 64 registers just to hold the CR coefficients.

To study the impact of enhanced ILP on modulo scheduling, Table 5 lists the static statistics extracted from the Sun Studio compiler for optimizing the computational loop with modulo scheduling in the three optimized versions of the *Poly3* code. The statistics are similar for the other benchmarks. The steady-state cycle count determines the performance, where lower cycle counts indicate that fewer cycles are executed per loop iteration, where each cycle issues at most 4 instructions. The observed performance is indeed best for *Poly3 VCR4.1*.

Figure 12: *Poly3* Benchmark Performance Results of VCR-Enhanced ILP (UltraSPARC IIIi 1.2GHz)

To investigate the scalability of the performance the ILP-enhancement of the *VCR4.1* code over interval $[10, \dots, 10^6]$, we plotted the results in Figure 12 for the *Poly3* benchmark. The steady-state cycle count (Table 5) is clearly reflected in the performance differences with *VCR4.1* having the lowest steady-state cycle count (3) and highest performance (1.0 GFLOPS peak).

Overall, the VCR approach significantly increases the ILP by decoupling factors $d \geq 1$. This benefits the performance of modulo scheduling for superscalar RISC architectures, as long as sufficient registers are available to hold the VCR coefficients. The number of registers needed for the VCR computation is dk , where k is the length of the CR form constructed for a function.

4. RELATED WORK

The CR formalism was originally developed by Zima [18] and applied to computer algebra systems [17]. The formalism was improved by Bachmann, Zima, and Wang [6, 7] to expedite the evaluation of multivariate functions on regular grids. Multivariate CR forms (MCR) are nested CR forms that represent multivariate closed-form functions over sets of index variables. Van Engelen [12, 13] extended the CR algebra by incorporating new rules and techniques for induction variable detection and optimization in compilers.

Zima et al. [20] investigated parallel mappings to evaluate CR forms using coarse-grain thread-level parallelism. A data partitioning approach is proposed to divide the iteration domain into p sub-domains given p processors to speed up execution and reduce the error. Several thread-level parallel execution strategies (functional parallel, data parallel, and subdomain parallel) are compared for two CR forms. By contrast, our approach proposes a decoupling strategy to exploit fine-grain parallelism by using a CR translation technique to generate independent value sequences, where the sequences are stored and updated in vector registers. Our technique can be combined with thread-level parallelism of the outer block loop (Figure 1) to further increase the execution speed of math function evaluations over grids.

Zima et al. [20] concluded that the performance of “functional parallel” CR evaluation is poor using thread-level parallelism. In our work, despite reducing the overhead of

“functional parallel” execution of CR forms, we also found that the resulting shuffle-based CR vectorization is not competitive due to the overhead of the tightly-coupled copy, shuffle, and add (or multiply) operations required for each grid point.

The CORDIC [15] family of algorithms provide a fast method to evaluate transcendental functions, roots, logs, and exponents for a single point. The CORDIC algorithm performs a rotation using a series of specific incremental rotation angles to approach the target angle and each step only requires addition, subtraction, bit-shift and table lookup. It is widely used in pocket calculators and real-time systems since CORDIC is generally faster than other approaches when a hardware multiplier is not available, or when the cost of the chip need to be minimized. The Intel 80x87 coprocessor series until Intel 80486 all use CORDIC algorithms [16]. These hardware advances are instrumental to reduce latencies that would otherwise diminish the effectiveness of our vectorization method.

Vector math libraries, such as Intel’s SVML and the Vector Math Library (VML) are highly optimized for SIMD short vector execution [9]. SVML is developed to provide efficient software implementations for transcendental functions on packed floating-point numbers with fully-accuracy [8] and is only intended for use by the Intel compiler vectorizer. VML is an application level library designed to compute math functions on vector arguments [2].

By contrast to these highly optimized vector math libraries, VCR vectorization is not restricted to math functions alone. A floating point expression composed of multiple math functions can be transformed to a VCR and then optimized together, resulting in more efficient evaluation.

5. CONCLUSIONS AND FUTURE WORK

In this paper we proposed a SIMD vectorization method to speed up evaluation of floating-point functions in loops. Many applications that require repeated evaluation of functions over regular and structured grids can benefit from the performance increase obtained by this method, such as plotting in a coordinate system, rendering of parametric objects in 2D/3D, numerical grid generation, and signal processing. A prototype automatic code generator is discussed that generates C code with SSE intrinsics to speed up the execution of functions represented in vector chains of recurrence forms.

Experimental results show that a dramatic performance increase is obtained compared to the best vectorizers on an Intel Xeon processor, from doubling the execution speed to running an order of magnitude faster. Furthermore, the effectiveness of modulo scheduling of function evaluations in loops is improved by the proposed technique, which results in faster kernels on superscalar RISC machines, such as the UltraSPARC IIIi.

The main contributions of this paper can be summarized as follows:

- The present CR formalism to expedite function evaluation over regular grids requires independent operations across the iteration space, which prevents loop vectorization. A new fine-grain decoupling strategy is proposed to vectorize CR evaluation for efficient SIMD vector execution and enhanced ILP. The CR-based method is applicable to any function defined over a commutative ring $(Z, +, *)$.

- A systematic performance evaluation of the proposed vectorization method is conducted for polynomial and non-polynomial (exponential) classes of functions in the real and complex domains and compared to existing work.
- A systematic performance evaluation of ILP-enhancing properties of the method is conducted and the impact on modulo scheduling is analyzed.
- An analysis overview of roundoff error properties of CR forms is given, with a motivation for the decoupling strategy to slow the rate of error propagation.
- An auto-tuning approach is proposed to determine optimal decoupling factor and block size for vector CR execution. This ensures optimality of performance and floating point accuracy.

In [11, 12, 13] we developed several compiler analysis and loop optimization techniques based on CR forms, such as nonlinear induction variable recognition, pointer-to-array access conversion, and nonlinear array data dependence analysis. These methods rely on the detection of induction variable recurrences from code. In our future work we will consider combining these loop analysis methods with the technique proposed in this paper to implement aggressive loop strength reduction with vectorization. However, the compiler optimization techniques will be limited to integer value sequences to ensure safety of these optimizations.

Further development will be conducted to further improve our prototype implementation to release a production-quality tool for vector CR generation and optimization.

6. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their constructive comments and suggestions that helped improve the content of this paper.

7. REFERENCES

- [1] SWI-Prolog's Home. Available from <http://www.swi-prolog.org/>.
- [2] Intel Math Kernel Library Reference Manual, March 2007. Intel Document number: 630813-025US.
- [3] A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Company, Reading MA, 1985.
- [4] V. H. Allan, R. B. Jones, R. M. Lee and S. J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.
- [5] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
- [6] O. Bachmann. *Chains of Recurrences*. PhD thesis, Kent State University, College of Arts and Sciences, 1996.
- [7] O. Bachmann, P. Wang and E. Zima. Chains of Recurrences - a Method to Expedite the Evaluation of Closed-Form Functions. In *proceedings of the International Symposium on Symbolic and Algebraic Computing (ISSAC)*, pages 242-249, Oxford, 1994. ACM.
- [8] A. J. C. Bik. *Software Vectorization Handbook, The: Applying Intel Multimedia Extensions for Maximum Performance*. Intel Press, 2004.
- [9] L. Kyle. How to Avoid Bottlenecks in Simple Math Functions. Available from <http://softwarecommunity.intel.com/articles/eng/3524.htm>, Intel 2004.
- [10] R. van Engelen. Symbolic Evaluation of Chains of Recurrences for Loop Optimization. Technical report, TR-000102, Computer Science Dept., Florida State University, 2000.
- [11] R. van Engelen. Efficient Symbolic Analysis for Optimizing Compilers. In *proceedings of the ETAPS Conference on Compiler Construction 2001, LNCS 2027*, pages 118-132, 2001.
- [12] R. van Engelen, J. Birch, Y. Shou, B. Walsh and K. Gallivan. A Unified Framework for Nonlinear Dependence Testing and Symbolic Analysis. In *proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 106-115, 2004.
- [13] R. van Engelen and K. Gallivan. An Efficient Algorithm for Pointer-to-Array Access Conversion for Compiling and Optimizing DSP Applications. In *proceedings of the International Workshop on Innovative Architectures for Future Generation High-Performance Processors and Systems (IWIA) 2001*, pages 80-89, Maui, Hawaii, 2001.
- [14] R. van Engelen, L. Wolters and G. Cats. CTADEL: A Generator of Multi-Platform High Performance Codes for PDE-based Scientific Applications. In *10th ACM International Conference on Supercomputing*, pages 86-93, New York, 1996. ACM Press.
- [15] J. E. Volder. The Cordic Trigonometric Computing Technique. *IRE Transactions Electronic Computers*, EC-8:330-334, September 1959.
- [16] A. K. Yuen. Intel's Floating-Point Processors. In *Electro/88 Conference Record*, pages 48/5/1-7, 1988.
- [17] E. V. Zima. Recurrent Relations and Speed-up of Computations using Computer Algebra Systems. In *proceedings of DISCO'92*, pages 152-161. LNCS 721, 1992.
- [18] E. V. Zima. Automatic construction of systems of recurrence relations. *USSR Computational Mathematics and Mathematical Physics*, 24(11-12):193-197, 1986.
- [19] E. V. Zima. On computational properties of chains of recurrences. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, page 345. ACM Press, 2001.
- [20] E. V. Zima, K. R. Vadivelu and T. L. Casavant. Mapping Techniques for Parallel Evaluation of Chains of Recurrences. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 620–624, Washington, DC, USA, 1996. IEEE Computer Society.