

Multidimensional Dataflow-based Parallelization for Multimedia Instruction Set Extensions

Lewis B. Baumstark, Jr.
Department of Computer Science
University of West Georgia
lewisb@westga.edu

Linda M. Wills
Dept. of Electrical and Computer Engineering
Georgia Institute of Technology
linda.wills@ece.gatech.edu

Abstract

In retargeting loop-based code for multimedia instruction set extensions, a critical issue is that vector data types of mixed precision within a loop body complicate the parallelization process since corresponding array elements are misaligned in the packed vectors. This paper presents a reverse-engineering approach to parallelization which extracts from the source code a multidimensional dataflow graph representation with explicit parallel semantics. The multidimensional annotations facilitate generating vector data type conversion code during code synthesis. This representation is independent of sequential artifacts, allowing code synthesis to proceed based on an abstract data-parallel model of the program and the constraints imposed by the architecture, such as vector length and available data types. Our results show that this representation facilitates parallelization of a wider range of loops than traditional vectorization. The results of this parallelization indicate loop speedups of 2 to 27 times over sequential execution.

1. Introduction

Multimedia software applications, such as video playback, image processing, and 3D video games, typically have a high degree of inherent data-level parallelism (DLP), since they often perform the same operation(s) over all the elements of a large data set. To harness this data parallelism, the instruction sets of general purpose and embedded processors have been extended with multimedia instructions that incorporate single-instruction, multiple-data (SIMD) functionality. Examples are Intel's SSE2 [1] and AMD's 3DNow! [2]. These subword parallel instructions simultaneously operate on multiple data elements packed into a single register, resulting in execution speedup. For example, the PADDW ("Packed Add Word") instruction included with Intel's SSE2 [1] extensions adds the corresponding elements of two

vectors of eight 16-bit data elements, providing an ideal speedup of 8 over a sequential execution.

However, developing and porting applications for these instruction sets has proven challenging. Typically, loop kernels employing multimedia instructions must be hand-optimized using in-lined assembly code or intrinsic functions. Traditional loop vectorization has been employed to generate SIMD code automatically, but is, in general, a transliteration technique that can only specify (to a back-end code generator) which loop statements can be executed in parallel. A deeper program understanding is required to deal with issues beyond loop-carried dependencies, such as architectural constraints. One such issue is that of misaligned data accesses resulting from mixed-precision data types; vectorization has no knowledge of how these data types interact. In this paper, we provide a program representation that explicitly encodes the array regions operated upon by the loop, allowing a code generator to recognize these misaligned data accesses and generate appropriate alignment code.

The main architectural constraint that must be considered when parallelizing loops for multimedia instruction sets is that all accesses to the elements of an array must be consecutive, i.e., having a stride of one. This follows from the parallel load and store operations common to these instruction sets where an entire vector of consecutive data is read from or written to memory in a single instruction (e.g., 16 consecutive bytes in SSE2 [1]).

A loop kernel that is not carefully written can easily violate this unit stride constraint. The simplest case is a loop where the array indices are monotonically increasing by a stride greater than one. We have shown in earlier work [9] that some loops can be restructured to have unit stride. However, a common case where unit stride becomes an issue is the presence of vector data types of different precisions within a loop body. For example, a loop having both 16- and 32-bit types will not align corresponding elements properly; in this case, the 16-bit vectors have

unit stride but the 32-bit vectors have a stride of two with respect to the smaller type. The compiler must deal with multiple parallel factors (e.g., 8-way and 4-way parallelism for the above types, assuming a 128-bit SIMD register size) in the same loop.

This research addresses the mixed data-type issue (and, as shown in [9], the non-unit stride issue), increasing the range of loops that can be optimized using multimedia instruction sets. Specifically, we take a reverse-engineering approach in which data-parallel access patterns are recognized in the program. These are abstracted to an explicitly-parallel, multidimensional dataflow (MDDF) graph-based representation of the program. The MDDF representation explicitly encodes array region annotations that facilitate resolving differing vector data types along graph edges. We developed a system, called PARRET (for PARallel RETargeter), for extracting MDDF from sequential source code and, from that representation, synthesizing data-parallel code. The MDDF representation abstracts out the sequential details used to implement the algorithm as well as architectural details specific to particular hardware. After extracting the MDDF specification, PARRET generates the appropriate subword parallel code for multimedia ISAs, using knowledge of array regions formed by each vector data type to generate type-conversion code to properly align mixed-precision vector types. The back-end code generator attempts to find best-case parallelism at a platform-independent level, for example, using knowledge of loop nesting to normalize loops with non-unit stride into unit stride when possible.

The MDDF representation also facilitates multi-target code generation. We have previously demonstrated PARRET's ability to generate code for SIMD processor arrays where parallelism is achieved via independent processing units [9]. This paper demonstrates its applicability to multimedia instruction set extensions that must instead deal with packed data issues such as data alignment and mixed data types.

We show that this method is capable of parallelizing a range of loop kernels beyond that of traditional vectorization. We identified a set of production programs that exhibit parallelization challenges, particularly in misaligned mixed-precision data types. These are nine programs from the Texas Instruments (TI) IMGLIB [13] suite for the TI TMS320C62xx line of DSPs; all nine were successfully parallelized by PARRET while only two were parallelized by a vectorizing compiler. Retargeting produces an average speedup of 2 over sequential across our test suite, with a maximum of 27.

The remainder of this section presents related work on vectorization to parallelize program loops for

multimedia ISA extensions. Section 2 gives an overview of MDDF, PARRET's recognition process, and the synthesis of code for multimedia ISAs from MDDF. Section 3 validates the correctness of programs retargeted to Intel's SSE2 [1] by PARRET, compares their speedup over sequential, and compares the ability of PARRET to parallelize loops with that of ICL, Intel's commercial vectorizing compiler. The last section presents conclusions and discussion.

1.1. Related Work

The traditional method for parallelization of loops is vectorization, i.e., partitioning the iteration space and data set of a loop such that each partition can be executed on parallel hardware. The main constraint for vectorization is that loops not have any loop-carried dependencies [5]. Once a vectorizing compiler is satisfied that dependence constraints have been met, it can schedule concurrent iterations for parallel execution based on the hardware resources available.

1.1.1. Vectorization for Multimedia ISAs. Several projects have implemented vectorization for generating code using multimedia ISAs, including the Intel C/C++ compiler [4], the MOM project [6], and work such as that by Sreraman *et al.* [7]. In general, these projects attempt to vectorize the inner loop of a loop nest. First, loop dependency analysis is performed. Then any applicable loop transforms are applied, such as loop distribution to isolate dependent statements into separate loops and scalar expansion (distributing the value of a scalar to an array that replaces the scalar) to break unnecessary loop-carried dependencies. Finally, if the loop is determined to be vectorizable, it is stripped to vector length, i.e., the loop step is changed from one to the vector length, such that each iteration operates on a single vector [8]. Code generation usually produces inlined assembly instructions from the multimedia ISA extension (or function calls that execute those instructions). Some, like the Intel compiler, perform low-level pattern recognition to identify code fragments, such as reduction operations (e.g., summation, minimum/maximum, etc.), that contain true dependencies, but that can still be parallelized when understood as the higher-level operation.

The Matrix-Oriented Multimedia (MOM) project [6] takes a unique approach to both multimedia ISAs and their compiler support. The MOM ISA provides two-dimensional, i.e., matrix, packed data types. These matrix types are modeled as a vector (the matrix rows) of packed words (the elements in each row). The MOM compiler builds these matrices by extending vectorization into two-dimensions. First, the inner

loop (of a doubly-nested loop) is vectorized in the usual manner. Then the outer loop is itself vectorized, using the results of the first step, to produce the matrices as vectors of vectors.

The main limitation of these types of compilers is their dependence on the syntactic details of the program. Compiler intermediate representations (IRs, e.g., abstract syntax trees, data-dependence graphs, triples/quads, register-transfer language, control-flow graphs, static single-assignment, etc.) are tightly coupled to the sequential details of the program (use of variables, loops, assignment statements, etc.) and as such still follow an inherently sequential computational model. They use explicit array accesses to determine loop-carried dependencies (or lack thereof) with the intention of strip-mining the loop to a specific vector length (i.e., changing the original loop stride to be the vector length). Here, the strip-mining process provides some basic understanding of the array regions, but only for a single vector length. However, in the presence of mixed-precision data types, this single-vector-length creates vectors of varying bit-width, violating the fixed-width constraint for multimedia registers, and the vectorization-and-strip-mining combination falls short. Our program representation encodes all region annotations separately and represents compositions of the region operations with dataflow arcs. This exposes the interaction between the operations on mixed-precision types and facilitates the generation of code that properly aligns elements between mixed-precision arrays.

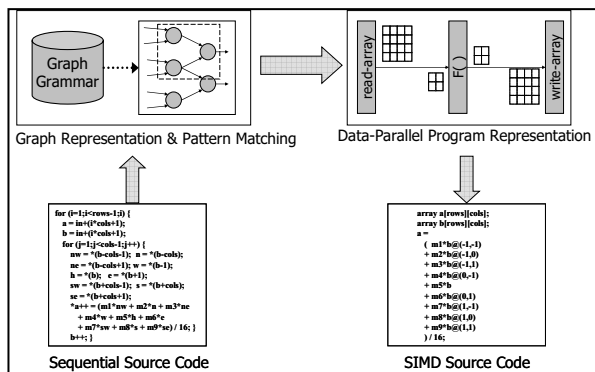


Fig. 1. Retargeting process.

1.1.2. Data alignment optimizations. An issue related to mixed-precision data alignment is the alignment of memory addresses for vector load and store operations for efficient access. For example, in SSE2, 128-bit vector data aligned to 16-byte boundaries can be accessed with the MOVDQA memory instruction much faster than with the unaligned MOVDBQ [1]; it is desirable that a compiler use the aligned instructions as often as possible.

Larsen *et al.* [3] have addressed this via memory congruence analysis. Bik *et al.* [4] address this by loop peeling, i.e., running sequential iterations of the loop until an aligned address is found, at which point execution of an aligned and vectorized version of the loop commences. These techniques are complementary to our work: our analysis of mixed-precision data types would generate load and store code to resolve the precision issues, after which that code could be further optimized using the above techniques in order to align those accesses at (e.g., in SSE2) 16-byte boundaries.

2. Approach

Fig. 1 illustrates the retargeting process. Our recent work [9] has investigated using recognition-based reverse engineering techniques to identify data-parallel memory access and computation patterns in sequential source code. Once identified, these patterns are used to create a high-level program representation with explicitly data-parallel semantics, from which data-parallel source code can be synthesized. The remainder of this section will give a brief overview.

2.1. Extracting an Explicitly-Parallel Program Representation

2.1.1. Target Representation. The goal in recognizing data-parallel patterns in source code is to create a high-level, explicitly parallel program representation. We chose a representation based on the multidimensional synchronous dataflow (MDSDF) [10] model of computation (MOC). MDSDF specifies a task-concurrency model where tasks, represented as nodes in a flow graph, communicate by sending tokens along the graph edges. These tokens are multidimensional entities where each dimension has a known integer size, e.g., a particular token may have three rows and two columns. Additionally, the producer for an edge may produce a token with different sizes and dimensions than the consumer of a node, resulting in differing rates of execution for each task. For example, if task T_1 produces a 4×2 token onto edge $E_{1,2}$ and task T_2 consumes a 2×1 token from $E_{1,2}$, then T_2 must execute four times for every execution of T_1 . These execution rates can be used by a task scheduler to create a static execution schedule for all tasks in the system.

Even though our domain is data-parallel program representation and not task-concurrency, the MDSDF model comes very close to providing the required data-parallel interpretation. If the multidimensional tokens are viewed as regions of array data, they become a natural way to express the data structures common to multimedia applications, such as images and audio

streams, as well as the partitions of these data structures that are often operated upon in isolation, e.g., columns and rows of images, sub-blocks of images and data-streams, etc. For example, consider the abstract algorithm representation in Fig. 2. This algorithm reads in a 512×512 -pixel image and a 1×128 -element coefficient array, partitions the image into 1×128 -pixel sub-blocks, multiplies the sub-blocks element-wise with the elements of the coefficient array, and then re-assembles the resulting blocks into the 512×512 -pixel output array. With the program in this representation, data-parallel code can be generated such that the multiplication operation is performed in parallel on SIMD hardware.

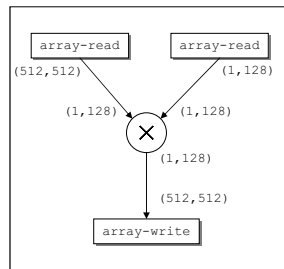


Fig. 2. Sample image-processing algorithm with a MDSDF-like representation.

We refer to our representation as *multidimensional dataflow* (MDDF) since we do not use it for synchronous task scheduling. We refer to the edge annotations describing the tokens as *regions*, similar to the notion of array regions described in [11], and notate our regions as $(start_y:range_y:step_y, start_x:range_x:step_x)$, where $start_i$, $range_i$, and $step_i$ refer to the first index of the region, the number of elements in the region, and the stride between elements, respectively, in each dimension i . For one-dimensional regions, the y -fields are removed for convenience, e.g., $(start_x:range_x:step_x)$.

2.1.2. Pattern Recognition. The recognition begins by performing control- and data-flow analyses on the source code, producing a dataflow graph-based intermediate form. For example, the loop of Fig. 3, a Sobel edge-detection algorithm from the TI IMGLIB suite [13], is converted into the intermediate dataflow graph (DFG) of Fig. 4. For brevity, portions of the code are elided. Note this example points out the mixed vector data type problem discussed in the introduction (see annotations in Fig. 4). Our previous work [9] presents an example where the loop stride is not one, but where PARRET is able to recognize from the loop nesting structure that the program is operating on contiguous data with a stride of one.

```

01:   unsigned char * in;
02:   unsigned char * out;
03:   short cols, rows;
04:   int H, O, V, i, j;
05:   int i00=0, i01=0, i02=0;
06:   int i10=0,      i12=0;
07:   int i20=0, i21=0, i22=0;
08:   int w = cols;
09:   for(i=0; i<cols*(rows-2)-2; i++){
10:     i00=in[i      ];
11:     i01=in[i + 1  ];
12:     i02=in[i + 2  ];
13:     i10=in[i+ w  ];
14:     i12=in[i+ w+2];
15:     i20=in[i+2*w  ];
16:     i21=in[i+2*w+1];
17:     i22=in[i+2*w+2];
18:     H= -i00-2*i01-i02+
19:     +i20+2*i21+i22;
20:     V= -i00+i02
21:     -2*i10+2*i12
22:     -i20+i22;
23:     A = abs(H) + abs(V);
24:     if (A > 255) A = 255;
25:     out[i + 1] = A;
26:   }

```

Fig. 3. Sobel example source code.

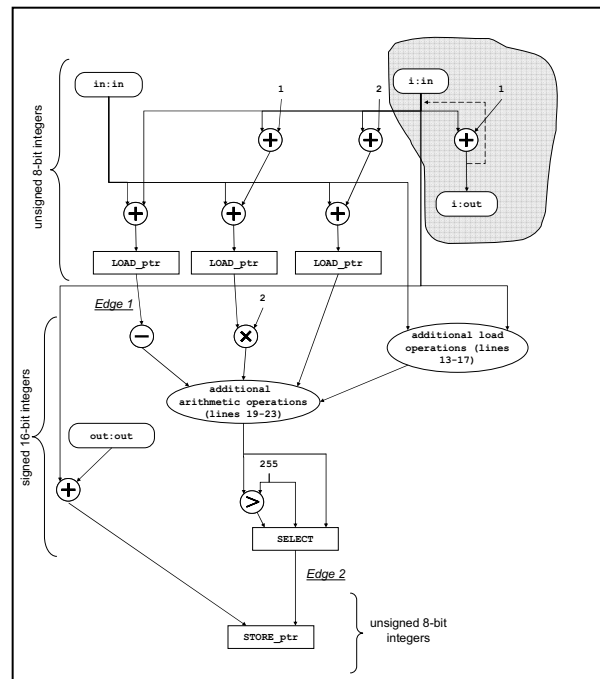


Fig. 4. Intermediate DFG representation.

Once the intermediate representation has been built, pattern matching can begin. The recognizer, using a graph-matching technique (see [9]), searches for instances of data-parallel patterns. Example patterns are shown in Fig. 5; a full description of the pattern library is given in [12]. Fig. 6 shows the recognition of a *count* pattern (a function that generates a linear sequence of values) from the cyclic structure created by the induction variable i . This corresponds to the shaded region in Fig. 4. Fig. 7 then

shows the recognition of parallel input and output primitives (array-read and array-write, respectively) from the memory access subgraphs (the `LOAD_ptr` and `STORE_ptr` blocks with their address calculation inputs) and the count pattern (see shaded regions in Fig. 6), producing the final MDDF specification for this loop.

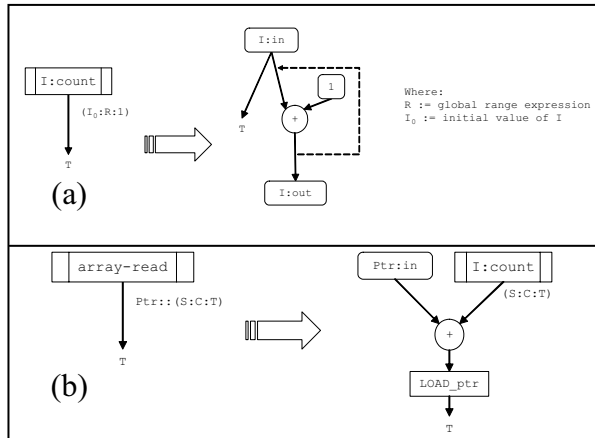


Fig. 5. Pattern examples: (a) Count pattern, (b) array-read pattern based on a count pattern and LOAD operation.

In this example, the `SELECT` node (and every other computational node, i.e. those not involved in memory address calculations) persists from the original DFG intermediate representation. Its semantics are that if the first input is a “true” value (non-zero value in C) the second input is quoted to the output, otherwise the third input is quoted to the output. This operator extends to a data-parallel domain by making arrays of all inputs and its output, applying the basic `SELECT` operation to corresponding elements of its inputs. This operation is equivalent to the bit-masking transformations used in mainstream vectorizing compilers [4][7].

This example illustrates PARRET’s knowledge of mixed vector data types and how they can be resolved. Consider the edge labeled *Edge 1* on Fig. 4 and Fig. 7. We chose the data type of the loop body calculations to be 16-bits (twice that of the input data, to provide adequate precision for the multiplication operations) and signed (because of the subtractions and negation). This means that the head of *Edge 1* is 8-bit unsigned (the input type) and its tail is 16-bit signed. In a multimedia ISA (assuming 128-bit vector registers), the head type would have 16 elements and the tail type would have 8 elements. The semantics of the representation dictate that two instances of the node supplying data to *Edge 1* (the `array-read` node) would have to execute for each instance of the node

that receives data from *Edge 1* (the negation node). A back-end code generator uses these relative execution ratios to generate the appropriate number of instances of data-parallel instructions for each operation.

2.1. Code Synthesis

Intel’s Streaming SIMD Extensions 2 (SSE2) [1] was chosen for analysis as a representative ISA with SIMD extensions. SSE2 is implemented in the Pentium 4 generation of x86 processors, facilitating analysis on commodity desktop workstations. It is supported at the assembly level by Microsoft’s Visual Studio, and at the source level, via vectorization, by Intel’s C/C++ Compiler.

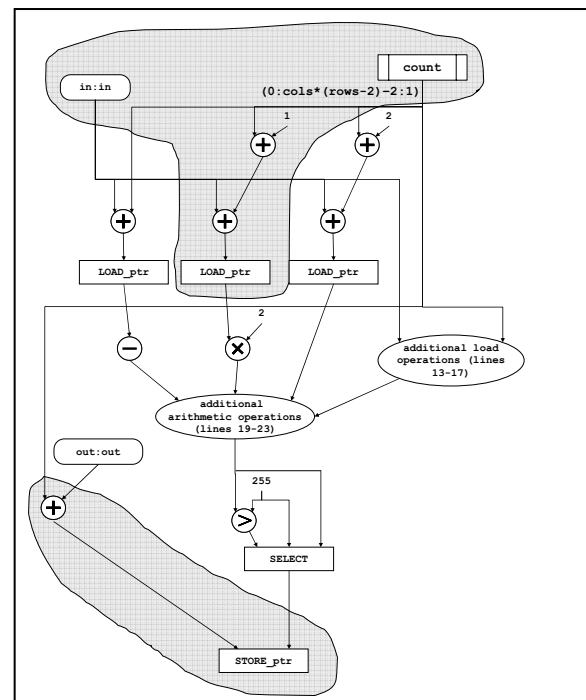


Fig. 6. Recognition of a count pattern.

Our approach to code generation for SSE2, particularly in the presence of mixed data types, is to generate code based on the highest-precision type. This could mean, for instance, that we load in 8-bit unsigned data, promote to 16-bit unsigned to perform higher-precision calculations such as multiplication, and then demote back to 8-bit unsigned for write-back, as must happen for the example of Fig. 3 through Fig. 7. The main constraint imposed by the architecture is that the parallel loads and stores must have the data packed consecutively, e.g., if all internal calculations are performed at 16-bit precision and the original program specifies output to an array of 32-bit integers, datatype promotion must occur before write-back.

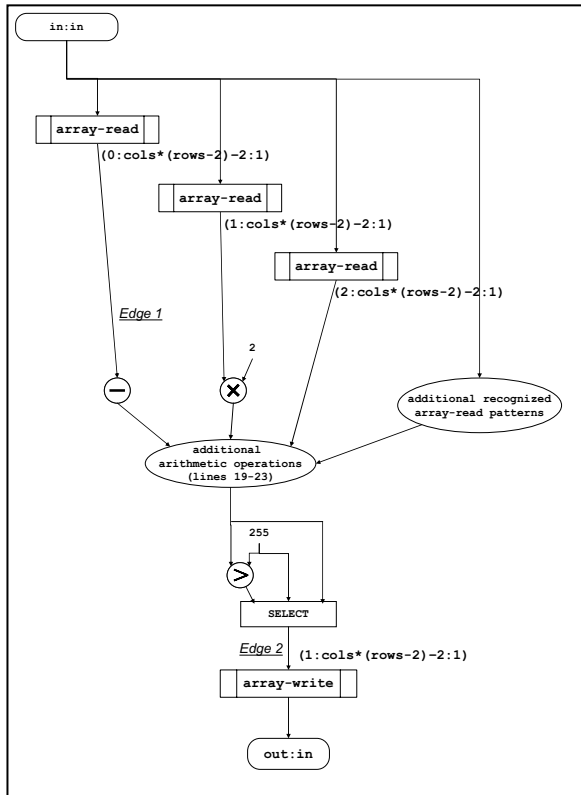


Fig. 7. Recognition of array-read and array-write patterns for finalized MDDF representation.

For brevity, the full details of code synthesis are not shown in this paper. A more thorough discussion can be found in [12]. The C code generated by PARRET for the sobel example can be obtained from http://www.westga.edu/~lewisb/sobel_parret.c.

3. Results

This section presents our experiments retargeting loops with inherent DLP to SSE2, a representative multimedia instruction set with SIMD functionality. We show PARRET is capable of synthesizing optimized code for SSE2 from the architecture-independent MDDF program representation. Additionally, we compare PARRET with a commercial vectorizing compiler and show that PARRET is capable of parallelizing programs the compiler cannot.

As a baseline for comparison, we chose Intel's C/C++ Compiler (ICL), version 8.0. It employs traditional loop vectorization to generate data-parallel code as well as providing a set of intrinsic functions for inlining SIMD operations in C/C++ source code. Our analysis will make use of ICL's vectorizing capabilities in characterizing whether or not applications can be

parallelized using traditional methods. The intrinsic functions provide a convenient set of primitives for PARRET to use when generating its own data-parallel code.

Table I lists the loop-based applications in our test suite. These are taken from the TI IMGLIB [13] library, a suite of image processing applications originally written for the Texas Instruments TMS320C62xx line of DSPs.

Table I. IMGLIB test programs.

Program	Description: Application(s)
conv_3x3	Convolution: Noise removal, image smoothing
corr_3x3	Correlation: Motion estimation
perimeter	Compute object perimeter: Object detection/recognition
pix_sat	Saturate pixels: Compression (i.e., clip values to a certain bit precision)
quantize	Quantize pixels: Compression (e.g., used in JPEG)
sobel	Edge detection: Object detection/recognition
threshold	Threshold pixels: Image dilation/erosion, perimeter detection
fdct_8x8	Discrete cosine transform: Image compression (e.g., used in JPEG)
mad_16x16	Min. absolute difference: Video compression (e.g., MPEG)

3.1. Retargeting of Image Processing Programs to SSE2

Table II reports on the ability of PARRET and ICL to parallelize the test programs. For each of the programs, ICL was executed using the "/QxW /O3 /Qvec_report3" flags. These flags specify, respectively, Pentium 4 code generation (including vectorization for SSE2), speed optimizations, and detailed reporting on vectorization attempts.

Table II. Suite coverage test.

Benchmark	PARRET	ICL
conv_3x3	Yes	No
corr_3x3	Yes	No
perimeter	Yes	No
pix_sat	Yes	No
quantize	Yes	No
sobel	Yes	No
threshold	Yes	Yes
fdct_8x8	Yes	No
mad_16x16	Yes	Yes

3.2. Correctness Validation

To validate correctness, each of the programs in the test suite retargeted by PARRET was executed for a 256-by-256 pixel input image and the results compared with those of a baseline execution of the original program as compiled by ICL (without vectorizing). Average distance (Equation 1) was used to compare the matrix results of each trial (except for

mad_16x16, which has a scalar result); these results are listed in Table III. Here, lower values are better, i.e., the matrix results closely match.

$$avg_dist = \frac{1}{MN} \sum_{n=1}^N \sum_{m=1}^M |a_{nm} - b_{nm}| \quad (1)$$

The two non-zero values listed in Table III (*perimeter* and *sobel*) result from precision differences caused by the saturation arithmetic used in SSE2.

3.3. Performance Evaluation

The performance gains of using parallelized code generated by PARRET are shown in Fig. 8. These were calculated as the time taken to execute the sequential version (as compiled by ICL) one hundred times divided by the time taken to execute the PARRET version one hundred times. Execution time was measured using the `_ftime()` function available from the Microsoft C/C++ runtime library (required for the Microsoft Windows version of ICL). All execution time tests were performed on a Pentium 4 1.80 GHz with 512 MB of RAM running Windows 2000.

Table III. Comparison of computational results, retargeted vs. sequential.

Program	Average Distance
perimeter	0.11
corr_3x3	0
quantize	0
sobel	0.01
pix_sat	0
thr_le2thr	0
conv_3x3	0
fdct_8x8	0
mad_16x16	Scalar result, exact match

Of interest is the large speedup seen with the *quantize* and *mad_16x16* benchmarks. The *quantize* benchmark was originally written as an inner loop over sub-blocks of an array, and an outer loop over the elements of each sub-block, resulting in an inner loop stride greater than one. This caused an increase in cache miss overhead in the sequential version due to lower spatial locality. PARRET, however, was able to recognize the loop nest as a sequence of operations on stride-1 data and generate the appropriate optimized code. The high performance gains seen in the *mad_16x16* benchmark were a result of the loop structure that PARRET produced: it generated four copies each of 4-way parallel versions of the instructions needed to complete the calculations, effectively unrolling the loop entirely (the loop was written as a 16-iteration loop). This eliminated the loop overhead and several expressions based on the loop index (now a constant) would have been calculated at compile-time instead of run-time. The trade-off is a larger code size.

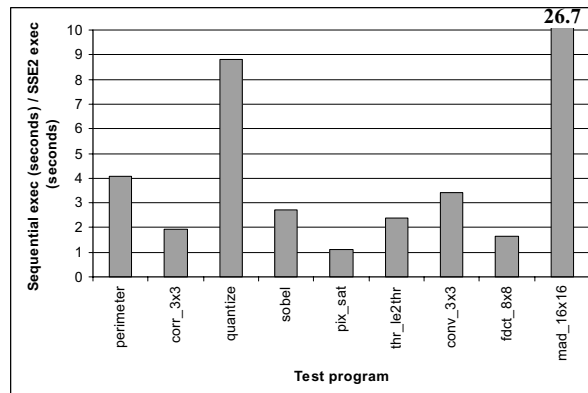


Fig. 8. Performance increase from retargeting.

Of the nine benchmarks that PARRET parallelized, there were only two that ICL was able to vectorize: *threshold* and *mad_16x16*. PARRET produced the same performance gain for *threshold* as the ICL vectorized version. However, the vectorized version of *mad_16x16* produced by ICL had a 6.5 times greater performance gain than PARRET's parallelized version. An examination of the generated assembly code revealed that ICL was able to recognize a sum of absolute differences operation in the original C loop and replace that code with the special-purpose instruction PSADBW ("packed sum of absolute differences"), saving several instructions that PARRET had to generate explicitly. In the future, such recognition could be incorporated into PARRET by expanding its recognition pattern library. In addition, the intrinsic functions PARRET uses for optimized C code generation can result in as many as seven assembly instructions for a single operation instead of one or two that might result if code generation were performed at the assembly level. We used these intrinsic functions for convenience only; future versions of PARRET could generate assembly code directly, producing a smaller, faster executable.

3.4. Code Size Evaluation

The code-generation method presented here was expected to increase code size due to the overhead of using pseudo-instructions, vector data type promotion and demotion, multiple instances of the same operation (to operate on separate parts of a vector), and the use of intrinsic functions. Table IV reports the increase in code size of each retargeted loop (not the overall application) caused by PARRET. The average increase of 4.3 times could be reduced by generating code at the assembly level and not by using the intrinsic functions.

As an illustration of code increase on a full application, we replaced the DCT in the *cjpeg*

implementation of JPEG encoding (available from <http://www.ijg.org/>) with the *fdct_8x8* kernel from the IMGLIB library. Using the original *fdct_8x8* code, it compiled to 135,168 bytes. Using the PARRET-retargeted code, it compiled to 139,264 bytes, an increase of 3.03%.

4. Conclusions and Discussion

PARRET is able to optimize loop-based code for multimedia ISAs beyond the capabilities of traditional vectorization, particularly in cases where mixed-precision vector data types must be aligned for vector operations in the loop. Those properties which make PARRET effective are:

- an abstract multidimensional dataflow representation of recognized algorithms,
- the ability to recognize more complex iteration spaces (e.g., the *quantize* benchmark) and normalize them into vectorizable form, and
- an explicit representation of how mixed data types compose.

Future work in retargeting data-parallel programs to multimedia ISAs includes improving code-generation. Greater performance of the parallelized code can be gained by avoiding the use of intrinsic functions that create extraneous instructions. In addition, some multimedia ISA extensions contain instructions for optimizing cache accesses, such as prefetch instructions; with its base of memory layout and access patterns, PARRET can leverage these instructions in the retargeted code. Finally, it would be interesting to apply PARRET to additional two-dimensional multimedia ISA targets, such as MOM [6] and CSI [14]. These would be good retargeting candidates for PARRET given its ability to recognize data parallelism in multiple dimensions.

Table IV. Code Increase for PARRET-retargeted programs.

Benchmark	PARRET size (bytes)	ICL size (bytes)	Ratio
conv 3x3	6291	1136	5.54
corr 3x3	6194	1379	4.49
perimeter	4352	995	4.37
pix sat	4695	1230	3.82
quantize	4731	1202	3.94
sobel	5595	1376	4.07
threshold	4518	1701	2.66
fdct 8x8	6915	1414	4.89
mad 16x16	5668	1155	4.91
Average			4.30
ICL options: -c -Drestrict= -QxW -O3			

This work was supported in part by NSF Grant CCR-0092552.

References

- [1] Intel IA-32 Architecture Software Developer's Manual, Intel Order Numbers 253666 and 2253667, 2004.
- [2] Stuart Oberman, Greg Favor, and Fred Weber, "AMD 3DNow! Technology: Architecture and Implementations," *IEEE Micro*, Vol. 19, No. 2, pp. 37-48, March/April 1999.
- [3] Samuel Larsen *et al*, "Increasing and Detecting Memory Address Congruence," In *Proc. of the 2002 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT'02)*, Charlottesville, VA, pp. 18-29, September 2002.
- [4] Aart Bik *et al*, "Automatic Intra-Register Vectorization for the Intel Architecture," *Int'l Journal of Parallel Programming*, Vol. 30, No. 2, pp. 65-98, April 2002.
- [5] Utpal Banerjee *et al*, "Automatic Program Parallelization," *Proc. of the IEEE*, vol. 81, no. 2, pp. 211-243, 1993.
- [6] J. Corbal, M. Valero, and R. Espasa, "Exploiting a New Level of DLP in Multimedia Applications," In *Proc. of the 32nd Annual Int. Symposium on Microarchitecture (MICRO 32)*, Haifa, Israel, pp. 72-79, November 1999.
- [7] N. Sreraman, R. Govindarajan, "A Vectorizing Compiler for Multimedia Extensions," *Int. Journal of Parallel Programming*, Vol. 28, No. 4, pp. 363-400, August.
- [8] Kathryn S. McKinley, "Automatic and Interactive Parallelization", Rice University, Houston, TX, Technical Report CRPC-TR92214, April 1992.
- [9] L. Baumstark and L. Wills, "Retargeting Sequential Image-Processing Programs for Data-Parallel Execution," *IEEE Trans. on Software Engineering*, Vol. 31, No. 2, pp. 116-136, February 2005.
- [10] Praveen K. Murthy and Edward A. Lee, "Multidimensional Synchronous Dataflow," *IEEE Trans. on Signal Processing*, Vol. 50, No. 8, pp. 2064-2079, August 2002.
- [11] Bradford L. Chamberlain *et al*, "ZPL: A Machine Independent Programming Language for Parallel Computers," *IEEE Trans. on Software Engineering*, Vol. 26, No. 3, pp. 197-211, March 2000.
- [12] Lewis Baumstark, "Extracting data-level parallelism from sequential programs for SIMD execution," doctoral dissertation, Georgia Institute of Technology, 2004, UMI Catalog No. AAT 3154911.
- [13] *TMS320C62x Image/Video Processing Library Programmer's Reference*, Texas Instruments Literature Number SPRU400, March 2000.
- [14] Ben Juurlink *et al*, "Implementation and Evaluation of the Complex Streamed Instruction Set," In *Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques (PACT '01)*, Barcelona, Spain, pp. 73-82, September 2001.