

Autovectorization for GCC compiler

YOU CAN USE INDUSTRY-STANDARD BENCHMARKS TO IMPROVE COMPILER PERFORMANCE.

You can't go out in public these days without seeing someone talking on the phone, listening to music, taking a digital photograph, or even watching a video. With all of these electronic devices available today, engineers are constantly seeking ways to make them smaller and more efficient. One way to accomplish this goal is to compress any data that must be stored or transmitted, but this task requires a powerful processor.

A common practice is to use a DSP to handle data compression and decompression and a general-purpose processor to handle everything else. The source code targeting the general-purpose processor is often written in C or C++, whereas the source code targeting the DSP is often written in assembler. But most OEMs would prefer not to have two or more processors in their products, so many contemporary, general-purpose processors include SIMD (single-instruction-multiple-data) capabilities. General-purpose processors with SIMD capabilities can offer improved performance when executing complex algorithms, such as those in portable-device applications, and can run general-purpose code, such as a Linux OS.

The basic premise behind a vector/SIMD unit is to allow the processor to simultaneously execute a mathematical operation on multiple pieces of data within special vector registers. The benefit is that a single instruction inside the CPU effects the parallel operation. To support more programmers writing in C, developers are improving the GCC (GNU Compiler Collection) compiler to more efficiently take advantage of general-purpose processors with these SIMD capabilities.

Executing industry-standard EEMBC (Embedded Microprocessor Benchmark Consortium, www.eembc.org) benchmarks on the IBM (www.ibm.com) PowerPC 970FX, compiled with the GCC compiler, illustrates the results of these compiler optimizations. Specifically, out-of-the-box EEMBC TeleBench scores for the 970FX PowerPC processor yielded more than 150% better results than those the consortium published more than a year ago using the same processor and platform and running at the same speed, but with a different compiler. You can attribute the improved scores to autovectorization and FDP (feedback-directed program restructuring).

APPLYING AUTOVECTORIZATION TO GCC

Autovectorization improves the performance of programs compiled with GCC for processors that have vector/SIMD capabilities. You can download the GCC from the GNU Web site, <http://gcc.gnu.org>. For example, the IBM 970FX processor VMX (vector/SIMD-multimedia-extensions) unit provides

32 quad-word, 128-bit vector registers that can hold different-sized elements, such as signed and unsigned bytes, half-words, full words, and single-precision floating-point numbers. The VMX unit is a complex design, so using it involves penalties. For example, loading the vector registers causes overhead, and restrictions exist, mainly in the alignment of the data, to read into these registers. Until now, it has been difficult for compilers to automatically recognize when performance benefits you accrue with VMX outweigh the overhead.

GCC is a versatile and readily available compiler and finds wide use in server, desktop, and embedded-system settings. Typically, its optimization technology lags many commercially available compilers. For now, however, GCC is among the leaders in the compiler industry for autovectorization technology. IBM engineers contributed much of this new technology while working on the autovectorization-branch version of GCC, which is available at <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>. These changes will be part of the official Version 4.3 release.

Autovectorization targets loops in the program that a vector/SIMD unit can parallelize. The first step is to analyze the loop to see whether it qualifies for parallelization. For analysis of the loop, autovectorization uses recently added loop-analysis features of GCC in the tree-SSA (static-single-assignment) facility, which is available at <http://gcc.gnu.org/projects/>

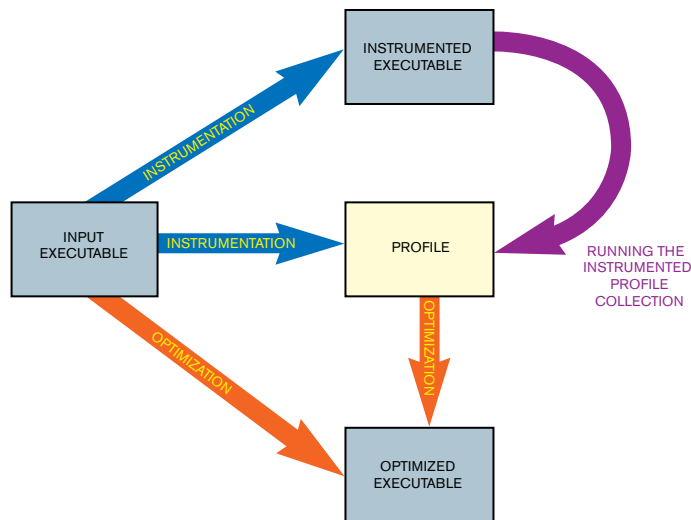


Figure 1 FDP-Pro takes an executable program as input.

tree-ssa. If it qualifies, GCC will convert the body of the innermost loop to a number of parallel operations, reducing the number of iterations by that factor. For example, if GCC can modify the loop to do two simultaneous operations, that modification would halve the number of iterations. Similarly, if the GCC can modify the loop to do four simultaneous operations, that modification would reduce the loop by a factor of four.

Architectural differences pose the main difficulty with autovectorization. Currently available vector/SIMD units typically use special-purpose instructions and special vector registers; they have many features that a general-purpose compiler would not use. Recently, developers have significantly improved the autovectorizer of GCC. Some of these improvements include the ability to handle “strided,” nonconsecutive accesses to memory; the addition of an idiom-recognition engine that includes a widening multiplication idiom; the ability to handle loops operating on multiple-sized data, including type conversion; the addition of enhanced “if” conversion and store sinking to replace conditional branches; and the addition of efficient loop versioning to resolve aliasing.

FEEDBACK-DIRECTED RESTRUCTURING

Another new technology to improve the performance on out-of-the-box code is FDPR-Pro. The FDPR-Pro postlink tool, available on the IBM alphaWorks Web site, www.alphaWorks.ibm.com/tech/fdprpro, analyzes the behavior of the compiled program when it is running a typical workload and creates an executable program after modifying the code and data layout to increase performance. FDPR-Pro takes an executable program as input (Figure 1). This executable program works in the same way as any other program, except for the addition of one extra flag that tells the linker to retain re-

TABLE 1 EEMBC OPTIMIZED-TELEMARK SCORES

Processor	Speed	Out-of-the-box score	Optimized-telemark score	Difference (%)
Freescale 7448	1.7 GHz	50.4	601.4	1093
Freescale 7447	1.3 GHz	37.7	507.8	1247
Freescale 7447	1.3 GHz	37.7	432.5	1047
Freescale 7447A	1.4 GHz	41.4	500.6	1109
Freescale 7455	1 GHz	28.3	121.6	330
IBM 970FX	2 GHz	56.1	1058.7	1787
LSI 402ZX	200 MHz	3.3	40.3	1121
TI TMS320C6203	300 MHz	6.8	44.6	556
TI TMS320C6203	300 MHz	6.8	68.5	907
TI TMS320C6413	500 MHz	13.5	263.3	1850
TI TMS320C6416	1 GHz	27.1	526.5	1843
TI TMS320C6416	1 GHz	27.1	873.1	3122
TI TMS320C6416	720 MHz	19.5	379.1	1844
TI TMS320C6416	720 MHz	19.5	628.6	3124
Average	NA	26.8	431.9	1512

location data; this extra flag results in a slightly larger file. The additional relocation information does not load to memory when the program is running.

FDPR-Pro first creates a new, instrumented, executable program by inserting additional code into the original program. This additional code is responsible for collecting profile information—typically, basic block- and control-flow edges’ execution counts.

Next, using a representative workload, it runs the instrumented program to create the profile data. Finally, it generates a new, optimized executable program, using the original program and the profile data.

FDPR-Pro can optimize any program or shared library by re-ordering the code to reduce cache and TLB (translation-look-aside-buffer) misses, reduce page faults and branch penalties, and improve branch prediction. It also removes unneeded NOP (no-operation) instructions, sets branch-prediction bits, and applies branch folding when beneficial. IBM engineers used FDPR-Pro to achieve an improvement of as much as 67% for single-kernel execution in the out-of-the-box EEMBC benchmark score for the 970FX.

BENEFITS OF GCC

Implementing effective autovectorizing optimization for

TABLE 2 RESULTS OF RUNNING EACH KERNEL IN TESTBENCH SUITE

Benchmark	GCC autovectorization	GCC autovectorization FDPR-Pro	FDPR gain (%)	GCC	Autovectorization gain (%)
Autocorrelation/pulse	2,264,526	2,285,207	1	2,649,007	-15
Autocorrelation/sine	114,943	117,647	2	23,669	386
Autocorrelation/speech	107,527	110,867	3	24,691	335
Convolutional encoder/xk5r2dt	696,864	698,813	0	48,309	1343
Convolutional encoder/xk4r2dt	700,525	763,505	9	54,644	1182
Convolutional encoder/xk3r2dt	1,028,560	1,096,191	7	63,694	1515
Fixed-point bit allocation/typical	11,835	12,063	2	10,000	18
Fixed-point bit allocation/step	110,538	184,502	67	163,044	-32
Fixed-point bit allocation/pent	17,410	17,718	2	15,000	16
FFT/IFFT/pulse	63,966	65,076	2	60,976	5
FFT/IFFT/spn	63,966	65,076	2	60,976	5
FFT/IFFT/sine	63,966	65,076	2	60,976	5
Viterbi decoder/get	40,236	41,459	3	9740	313
Viterbi decoder/toggle	40,226	41,448	3	10,638	278
Viterbi decoder/ones	40,237	41,460	3	11,450	251
Viterbi decoder/zeros	40,170	41,277	3	14,286	181
Telemark	133.8	141.8	6	49.2	172

a target processor is a difficult task. One of the many hurdles is the fact that test cases represent real programs. At this point, EEMBC telecommunication benchmarks come into play. IBM engineers running the EEMBC TeleBench kernels on the IBM PowerPC 970FX processor obtained data that they then used to improve the autovectorization capabilities of the GCC compiler. EEMBC's TeleBench suite comprises kernels that include autocorrelation, convolutional-encoder, bit-allocation, inverse-FFT (fast-Fourier-transform), FFT-benchmark, and Viterbi-decoder tests. These benchmarks represent tasks that can benefit from vector/SIMD execution. For example, the Viterbi encoder computes the most probable transmitted sequence of a convolutional coded sequence. The most computationally intensive part of Viterbi performs a maximization of a likelihood function through a sequence of add-compare-select operations.

EEMBC publishes two types of scores: out of the box and "full fury," or optimized. The organization obtains out-of-the-box scores by compiling unchanged source code, and it obtains full-fury scores by changing the source code to improve performance but still follow the EEMBC rules. In most cases, the changes engineers make to the code enable use of vector/SIMD instructions that compilers were unable to do automatically. The full-fury scores show an average improvement of more than 1500% over out-of-the-box scores for the same processor running at the same speed (**Table 1**).

You can make many optimizations by restructuring the code

MORE AT EDN.COM ▶

+ Go to www.edn.com/ms4249 and click on Feedback Loop to post a comment on this article.

or rewriting it in assembler, but a compiler that recognizes when using vector/SIMD will be beneficial could automatically do a significant portion of these gains. To illustrate this point, compare the IBM 970FX optimized score of 1058.7 to the out-of-the-box score of 56.1 in June 2005 using Green Hills Software's (www.ghs.com) Multi compiler (**Table 2**). The percentage of improvement with hand-optimizing is on the same order of magnitude as many of the other processors' optimized improvements. The out-of-the-box score when using GCC with autovectorization is 141.8. This score is 153% better than the previous out-of-the-box score. Internal testing at IBM shows that the compiler from Green Hills Software is better than the GCC compiler without autovectorization. The tests also show that, when comparing results of autovectorization with results compiled using the previous version of GCC, Version 4.1.1, the gain is closer to 190%.

The first column of **Table 2** shows the EEMBC score in iterations per second of the test EEMBC compiled with autovectorization but before running FDPR-Pro. The next two columns show the results of running FDPR-Pro on the autovectorized executable and the resulting performance improvement expressed as a percentage. The next two columns compare the results of compiling the benchmarks with autovectorization and without autovectorization, ignoring gains from FDPR-Pro in both cases.

As you can see from the "autovectorization-gain" column, autovectorization provided more of a boost to some of

the benchmarks, particularly Viterbi benchmarks. For other benchmarks, such as convolutional encoder, scalar replacement of aggregates, rather than autovectorization, improved scores. Vectorization can further boost the convolutional encoder, as the optimized-telemark version demonstrates, but it is more difficult to automate.

The benefit of improved compiler optimizations is not higher benchmark scores, but better performance in real products with less time and effort. Today, OEMs are demanding more performance from embedded processors, and processor vendors are responding by adding execution units that can do simultaneous operations in parallel, such as vector/SIMD engines. Adding capabilities to the CPU will do no good unless the software can take advantage of these features. Until now, the best choice for taking advantage of vector/SIMD functions has been to modify generic code to use these special features of the processor. This method has many disadvantages, including the fact that it binds you to a specific implementation. Now, with GCC 4.3 and beyond, you can benefit from vector/SIMD by simply recompiling your code. **EDN**

REFERENCES

- 1 "Auto-vectorization in GCC," <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>.
- 2 "Feedback Directed Program Restructuring," www.haifa.il.ibm.com/projects/systems/cot/fdpr.
- 3 "PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual," Aug 22, 2005, www-3.ibm.com/chips/techlib/techlib.nsf/

techdocs/FBFA164F824370F987256D6A006F424D.

- 4 Nuzman, Dorit, Ira Rosen, and Ayal Zaks, "Auto-Vectorization of Interleaved Data for SIMD," PLDI 2006, June 12 to 14, 2004, Ottawa, ON, Canada, pg 132.
- 5 Nuzman, Dorit, and Ayal Zaks, "Autovectorization in GCC—two years later," GCC Developers' Summit 2006, June 28 to 30, 2006, Ottawa, ON, Canada.
- 6 Nuzman, Dorit, and Richard Henderson, "Multi-platform Auto-vectorization," Proceedings of the International Symposium on Code Generation and Optimization, March 26 to 29, 2006, New York.

AUTHORS' BIOGRAPHIES



Markus Levy is president of EEMBC, where he has worked for 10 years and where he is responsible for management, marketing, business development, and conference presentations. He has a bachelor's degree in electrical engineering from San Francisco State University, and his personal interests include exercising, mountain biking, skiing, and volunteering as a firefighter. You can reach him at markus@eembc.org.



Ron Olson is a software engineer at IBM, where he has worked for 23 years. He is responsible for PowerPC-software development, benchmarking, and enablement. He has a bachelor's degree in electrical engineering from DeVry University, and his personal interests include sailing, fishing, and swing dancing. You can reach him at ronolson@us.ibm.com.