Unrolling AltiVec, Part 2: Optimize code for SIMD processing

Tweak your C code, or dip into assembly

Peter Seebach (crankyuser@seebs.plethora.net)
Author
Freelance

16 March 2005

In this second article of a three-part series, Peter Seebach looks closer at AltiVec, the PowerPC SIMD unit. He explains further how you can effectively use AltiVec, discussing the choice between C and assembly, and shows some of the issues you'll face when trying to get the best performance out of an AltiVec processor.

The Motorola® AltiVec™ instruction set offers a variety of vector operations. Vendors and reviewers claim that AltiVec is a particularly efficiently designed set of vector operations, allowing programmers to get excellent performance and make the most use of the multiple execution units it provides. But how do *you* get this performance?

This article, the second in the *Unrolling AltiVec* series, gives an overview of performance considerations that will help you get started making the best possible use of an AltiVec processor. (If you're unfamiliar with the basics of AltiVec, read Part 1 of this series before you go any further.) The compiler may get some or all of the ideas outlined here on its own, but if you're careful, you can probably do a better job. Still, check for likely compiler options before you spend a lot of time hand-tweaking code. If you can get acceptable performance by setting a compiler flag, do it and be happy.

One word of advice, though. Before you spend a lot of time optimizing a process, remember that the first rule of optimization is to find out where the time is going in the first place. If you spend a couple of days carefully getting the best possible performance out of a routine that accounts for one percent of all the CPU time your program uses, you are probably wasting your time. Before you even start on optimizing (unless you're doing it for fun, or to learn about vectorizing code), you should take a profiler to your program and find out where the time is spent. (`simg4` and `simg5` are both good profilers; see Resources for links.)

Great processors don't always think alike

One thing to keep in mind when optimizing code for AltiVec is that different AltiVec-equipped processors have different capabilities. The previous article in this series briefly mentioned these differences; this article goes into more depth about the differences. In short, the "AltiVec unit" is not really just a single processing component: it's a number of different components, and in principle, all of them can be running at once.

What's in a name?

AltiVec is a trademarked term used by Motorola to define its implementation of the SIMD unit this series discusses. *VMX* was the original code name for this extension inside IBM®, but generic terms like *SIMD* or *vector processor* are now preferred in IBM documentation. Both SIMD units are referred to as *Velocity Engine* in Apple marketing materials. I'll be using AltiVec in this series because I like the sound of it.

The Motorola chips marketed by Apple under the *G4* name fall into two categories -- G4 (including model numbers 7400 and 7410) and G4+ (including model numbers 7450 and 7455) -- with slightly different AltiVec implementations. The chips marketed by Apple as the *G5* include IBM chips with the model numbers 970 and 970FX.

Up to a point, the goal of optimization is to try to get as many components as possible working at the same time. That's the rule of thumb. However, an algorithm that goes through elaborate contortions to try to ensure that a given unit gets used for something may spend more time setting up the calculation than it saves by using that unit.

Thus far in the history of AltiVec processors, newer processors have generally been able to run more instructions at once. This doesn't necessarily make them faster on a single instruction; in fact, on average, they will take a cycle or two longer to perform a given operation. The newer processors still perform better, thanks to increased clock speed.

More subtly, the performance hit that these extra cycles cause is mitigated by the ability to start more instructions in a given cycle. For instance, on an original G4, most permutation instructions can run in a single cycle, while on a G4+, they may take two. However, on the original G4, two vector operations can start on the same cycle only if at least one of them is a permute operation; otherwise, the second operation must be delayed at least one cycle. On the G4+, three instructions may be dispatched in a single cycle, going to any two (different) vector processing units. On the G5, only one arithmetic operation and one permutation may be dispatched simultaneously, but up to two other instructions, plus a branch, may be dispatched to other parts of the processor, including the vector load/store units.

Unfortunately, the net result of these differences is that a program that runs brilliantly on one AltiVec processor may stall frequently on another. When optimizing code for general production, you must test it (or simulate it) on all the available types of processors, and focus on general principles. On the other hand, if you have a guarantee that your code will only be running on a specific processor, you can take advantage of more detailed knowledge of its instruction timing and its dispatch capabilities and limitations. General optimization strategies will give you code that works pretty well everywhere. Even if you end up moving code written for one specific processor to another, the porting process is not particularly difficult.

Loop-de-loops

If your code doesn't have any loops, it probably can't be vectorized much. There's still some room for taking advantage of multiple execution units, but AltiVec probably won't particularly help you. SIMD instructions are primarily good for doing the same thing over and over, and that normally means loops. If you aren't doing something multiple times, it probably doesn't take long enough to be worth trying to optimize. So, you want to vectorize your loops. Even if you can't do that, you can improve performance by *unrolling* them. This will also make it easier to see whether there are any opportunities for vectorization.

Unrolling?

To *unroll* a loop is to make each pass through the loop do the work of two or more passes. The simple code example in Listing 1 illustrates this:

Listing 1. Unrolling a loop

```
int
rolled_sum(char bytes[16]) {
 int i;
 int sum  = 0;
 for (i = 0; i < 16; ++i) {
  sum += bytes[i];
 }
 return sum;
}

int
unrolled_sum(char bytes[16]) {
 int i;
 int sum[4] = 0;
 for (i = 0; i < 16; i += 4) {
  sum[0] += bytes[i + 0];
  sum[1] += bytes[i + 1];
  sum[2] += bytes[i + 2];
  sum[3] += bytes[i + 3];
 }
 return sum[0] + sum[1] + sum[2] + sum[3];
}
```

Both versions of this function add 16 values to `sum`. However, the second does so in a way that makes it easy to see that there are four similar operations happening in a row. This might make it easier for a compiler -- or a programmer -- to see the opportunity to vectorize the code.

When you've unrolled a loop a bit, you can often begin to see possible opportunities for vectorization. Sometimes, you won't immediately; it may seem that each phase of the calculation depends on the previous one, so you can't do multiple parts simultaneously. However, in some cases, these dependencies can be dealt with by splitting a process up into parts. Even if this requires a small amount of extra calculation, the speed benefits of vectorizing might pay for it.

Unrolling is often an optimization in and of itself, because branching is typically comparatively expensive. However, some of the work that goes into preparing an algorithm for vectorization could actually reduce performance. Don't abandon the effort just because the intermediate stage isn't working out. It can take a while to get an optimization correct.

Division of labor

Now that you've got a nicely unrolled loop, it's time to start looking for patterns in it. If every run through the loop involves adding two numbers together, you might benefit from putting those numbers in a pair of vectors, and adding them once.

Loading vectors, performing a single operation, and writing back to memory isn't all that good of a deal. What you ideally want is a series of operations that occurs on vectors, writing back to memory only at the end. The more of your loop's core logic you can perform in vectors, the more benefit you're getting, and the more likely you are to get enough payoff to cover the cost of the setup involved in loading to, and storing from, the vector registers.

There is another concern here, though. Depending on the processor you're using, you might face some limitations. For instance, on a G5, you can't dispatch two vector arithmetic operations in a single cycle, although you can combine a vector arithmetic operation with a permute. Don't overuse permutes just because you can launch them cheaply, though; moving data back and forth between the arithmetic and permutation units will cost you a cycle each way.

If you can arrange to interleave vector operations with ongoing non-vector operations, you will improve performance noticeably. Similarly, if you have to do a lot of loads and stores along with your arithmetic operations, interleave them if you can.

Unrolling even further

Now, let's say you have a vectorized version of some loop. You may find that it's not particularly efficient. In particular, if you have a frequently called function that does a small amount of vector math, you might find that performance actually drops to an abysmal level.

Here is where you may wish to unroll your loop again. AltiVec gives you 32 vector registers. That's enough that you may well be able to do two or more versions of essentially the same vectorized loop in a row. This gives you a substantial performance advantage. With a single vectorized pass through a loop, all the data must be loaded in, processed, and stored before the next phase can start. If you have two or three passes through the loop, using different sets of registers, the third phase may well have loaded all of its data before the first phase has finished computing, giving you a substantial boost in speed.

Often, getting data into or out of the processor is as much a bottleneck as any actual computation. This is another area where unrolling can help dramatically. The earlier you start loading data, the sooner you can start processing it. The G5's multiple load/store units improve this situation a lot, allowing you to start two loads, or a load and a store, in a single cycle.

In the particular case where a moderately vectorizable function is called many times, it is almost certainly worth building a version of the function that runs on four or more sets of data at once, essentially unrolling the loop. Apple gives a very good example of how this would work using the regular scalar floating point unit. For another look at loop unrolling to take advantage of AltiVec, read the two-part developerWorks series on TCP/IP checksum vectorization (see Resources).

Assembly and processor simulations

If your best efforts using the C interface aren't getting the results you think you should be getting, it might be time to consider writing your own assembly code. This is not a step to be undertaken lightly. Most obviously, if you make a mistake involving the `VRSAVE` register, you may cause your program to fail catastrophically!

This is a good time to stop and profile your code. It is easy to get bogged down optimizing code which isn't even consuming all that much processor time. Start by finding out where the processor time is being spent, and optimize that code.

If you've already got C code, a good starting place may be to compile the code to assembly and look at the resulting code. It might be immediately obvious to you what went wrong, and indeed, you might find that a simple change to your original source gives the compiler enough hints to do the right thing.

If that doesn't work, it's time to get down and dirty. You may want to use a processor simulator, such as `simg4` or `simg5` (see Resources), to try to find out *exactly* where your code is getting bogged down. What you're looking for are *stalls,* where an instruction must be delayed as it waits for the results of another instruction, or for some processor resource limit that has been reached. Try to eliminate dependencies, or separate dependent operations, as much as possible, by putting other code between them.

Other resources

This article has given something of an overview of what's involved in optimizing AltiVec code. You can find a great deal of additional information on Apple's developer pages. Apple clearly has a substantial interest in making sure that programmers targeting the Mac are getting the best possible performance out of AltiVec processors.

Compiler documentation is often full of tips, tricks, and tidbits about optimization techniques supported, or ways to give hints to the compiler about what you're trying to do. Read it carefully. Generally this documentation will come with the compiler itself. If you're using `gcc`, you may want to check out the new GCC Wiki (see Resources).

Most importantly, if you have a piece of code that really needs to be optimized this heavily, profile it, study it, and try to find out what's happening. Don't be afraid to rethink the algorithm from the top. The world is full of hand-tuned assembly for bubble sort.

If you find all this discussion a bit too theoretical, you'll want to tune in for the third part of the series. In it, you'll see a more detailed, real-world example of unrolling and optimizing a loop, showing how to put some of these principles into practice. Stay tuned!

Resources

- Check out the first part of this series (developerWorks, March 2005) .
- Read the PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual.
- SIMD has working groups involved with various SIMD extensions, including AltiVec, MMX, and others. Don't miss their AltiVec tutorial (in PDF format) by Ian Ollman.
- Apple's page about the Velocity Engine is a slightly buzzword-heavy description of the AltiVec variants used in Mac systems.
- Duff's Device is the most famous example of loop unrolling.
- Users looking for a non-Macintosh PowerPC might be interested in Pegasos.
- Motorola recently spun off its chipmaking division into a separate company called Freescale. The Freescale site also has a page about AltiVec.
- A previous two-part developerWorks article, "TCP/IP checksum vectorization using AltiVec," by Ayal Zaks, Dorit Naishlos, and Daniel Citron, discussed TCP checksum vectorization using AltiVec; start with Part 1 and Part 2.
- A discussion of throughput vs. latency, on Apple's site, is of particular interest.
- Apple provides detailed performance information about the G4, G4+, and G5.
- Work is being done on auto-vectorization in `gcc`.
- Crescent Bay Software sells software to automatically vectorize C code.
- Apple's page on performance tools gives links to a number of useful tools, including `simg4` and `simg5`.
- The GCC Wiki serves as a repository for information about `gcc`, with up-to-the minute reports on status, useful tips, and everything else you might want.
- IBM Senior Processor Architect Peter Sandon discusses vector processing in the G5 in this interview.
- "Save your code from meltdown using PowerPC instructions," Jonathan Rentzsch gets into the gritty detail of PowerPC assembly code (developerWorks, November 2004).
- For more on the joys and dangers of writing code that directly accesses memory, check out "Data alignment on PowerPC," Jonathan Rentzsch (developerWorks, February 2005).
- The world is full of hand-tuned assembly for bubble sort.
- Have experience you'd be willing to share with Power Architecture zone readers? Article submissions on all aspects of Power Architecture technology from authors inside and outside IBM are welcomed. Check out the Power Architecture author FAQ to learn more.
- Have a question or comment on this story, or on Power Architecture technology in general? Post it in the Power Architecture technical forum or send in a letter to the editors.
- The Power Architecture Community Newsletter includes full-length articles as well as recent news about members of the Power Architecture community and upcoming events of interest. Subscribe to the newsletter today!
- All things Power are chronicled in the developerWorks Power Architecture editors' blog, which is just one of many developerWorks blogs.
- Find more articles and resources on Power Architecture technology and all things related in the developerWorks Power Architecture technology content area.
- Download a IBM PowerPC 405 Evaluation Kit to demo a SoC in a simulated environment, or just to explore the fully licensed version of Power Architecture technology. This and other fine

Power Architecture-related downloads are listed in the developerWorks Power Architecture technology content area's downloads section.

About the author

Peter Seebach

Peter Seebach uses vector processing a lot, and is personally able to cook up to three eggs at once, making him something of an expert in the field.