Accelerating C-based Applications by Running Parallelized Code on Configurable Processors

Steve Casselman, CTO
DRC Computer Corp., Sunnyvale, Calif.

Most software today is written so that instructions are executed in sequence, and to speed up execution programmers have typically pushed the hardware designers to build processor with ever higher clock rates. That has given rise to heavily pipelined processors that operate at clock rates of 3 GHz and more. These processors also include architectural tricks such as large caches, and functions such as out-of-order execution to get the most out of every clock cycle.

However faster processors generate lots of heat and today, clock speeds have, for the most part, leveled off since the heat generated by the faster circuits ends up constraining the clock speeds. To continue the march towards ever-faster execution, hardware designers have switched from a single processor on a chip to dual, quad, and even more CPU cores on a single chip. The operating system can then allocate the processors to different applications, all running in parallel. The next level down from there is to find ways to parallelize the code running in each application and then run that parallelized code on multiple engines either within the CPU or in a companion coprocessor that is optimized to execute that particular segment of parallelized code.

The latest generation field programmable gate arrays (FPGAs) and the new open-standard Torrenza coprocessor interface on Opteron system platforms defined by Advanced Micro Devices, and the Intel QuickAssist Technology Acceleration Abstraction Layer (AAL) provide designers with the hardware portion of the parallelization goal. By downloading configurations into an FPGA tied to either a Torrenza or QuickAssist motherboard platform, designers can accelerate computationally-complex algorithms such as encryption, compression, search and sort, up to 1000X over a general-purpose processor. Also, DSP algorithms that need billions of integer or floating-point operations per second for image and audio processing, and much more, can readily be accelerated by an FPGA-based coprocessor.

The challenge now becomes how to find the code within your current application that can be parallelized, and then how to parallelize that code so that it can be executed on an array of computational elements configured in an FPGA or ASIC. Where does one start

with this analysis? One good starting point is to first profile the code to find the computationally-intensive portions of the code and then find ways to isolate the code so that it does not have many data dependencies. Once this code is isolated, you must find ways to optimize the code so that it can be executed on the resources available on the coprocessor. Optimizing the code to fit the architecture is difficult on architectures like graphics processing units and the Cell processor, where users are not given all the data for the device. FPGAs, on the other hand, allow you to define an optimized architecture on a case-by-case basis.

Next, examine the communication channels between the hardware and software. Make sure the hardware is not data starved (not enough data reaches the hardware to keep the hardware continually performing its computations). And at the same time, make sure the hardware can keep up with the pace that the software feeds data to the hardware. Finding the right balance is critical to smooth operation. If it takes longer to transmit the data to the accelerator than it takes the CPU to calculate the answer, then make sure the hardware design uses all memory accesses most efficiently, or perhaps even add another memory port to feed more data to the compute array.

For example, if you are accelerating a fast-Fourier transform (FFT) followed by a phase shift and then followed by an inverse FFT, in the software algorithm, you take the data out of a memory location, perform the FFT and then put data back into memory. Then the data comes out of the memory again and the algorithm executes the phase shift computations and places the data back into memory one more time. Lastly, the inverse FFT retrieves the data, executes the algorithm and the places the final result back in memory. With a few hardware and software tricks data reuse can dramatically improve performance by eliminating multiple store and access operations.

When performing numerical calculations, analyze the precision that you really need to implement – you will probably find that you won't always need the full precision of single or double-precision floating-point computations. A fixed-point multiplier and arithmetic unit combo usually requires less logic and thus more elements can be configured on an FPGA. The more elements, the more computations per cycle and the faster the algorithm can execute. In FPGAs, integer or fixed-point math can often run 10 to 100 times fast than floating-point computations.

So where to start to parallelize the algorithms? First, perform a data-flow analysis to understand how data has to move between the different logic and computational elements. Next, perform a latency analysis to try and determine where potential bottlenecks may occur and then find a balance between desired performance and the cost of implementing the design—to achieve a desired performance level, will you have to use PCI express or Hypertransport to provide high-bandwidth, low latency communication channels. Or, will you need a larger FPGA than initially planned to host more computational elements? How will the larger FPGA affect the cost of your solution?

Lastly, scour the literature for tricks that can accelerate computations—some tricks may actually be decades old but were impractical before the availability of multimegagate

FPGAs. Revisit old ideas with a fresh view. Consider different types of math for computations aside from basic integer and floating-point operations—perhaps logarithmic computations or math based on residue number systems, etc. could execute faster than standard integer or floating-point-based code.

In seismic processing software, wave migration algorithms often execute fast-Fourier transforms as part of the solution. The FFT technique approach usually takes the following sequence: FFT, custom phase shift, and then an inverse FFT. To execute this with software, data is taken out of memory to compute the FFT then the result is put back into the memory. Then the result data is taken out to do the phase shift and the new result put back into the memory. Lastly, the new result data is taken out so the inverse FFT can be computed and the final result is put back into memory.

In such a linear execution there are too many memory accesses and they slow down the execution. By using hardware to accelerate the algorithm, start taking the data out of memory and feed it to a pipeline that executes the FFT, feeds the result into the phase shifter, and the phase shifter directly feeds its result into the inverse FFT. The output of the inverse FFT is then put back into memory.

A typical application level profile for a wave migration algorithm might look like:

FFT	35%
Inverse FFT	35%
Phase Shift	25%
I/O	3%
misc	2%

Traditionally these algorithms are executed using floating-point math operations since the input data is gathered from 24-bit sensors and the output is often displayed as 24-bit color pixels. By implementing multiple floating-point units in an FPGA, the FPGA can outperform a CPU 5 to 1 on floating-point code but can deliver a 10x – 100x improvement on integer based code.

Next see if the code will fit into hardware...

A quick look around the web yields a wide variety of floating-point and integer FFT algorithms. Floating-point versions are larger and slower than the integer versions, but deliver more precise results. One of the examples, a hybrid approach, can compute a 2Kpoint FFT at a rate of 400 Msamples/s (http://www.andraka.com/V4_FP_fft.htm). This is 4.5 times faster than a 2.2 GHz dual core Opteron (http://www.fftw.org/speed/opteron-2.2GHz-64bit/). Three such FFT engines can fit in a Xilinx SX55, however for the example selected only two FFT engines are needed. To do the phase shift a CORDIC algorithm can be used since implementing such an algorithm does not consume many of the FPGA's logic elements--less than 1000 look-up tables. Furthermore, the computations can be pipelined to improve throughput.

At first you might think “ok so it goes 4.5 times faster--I can still buy a 4 way Opteron based system for the same price as an FPGA-based coprocessor and a dual-core Opteron.” But since the functional units will be pipelined, the phase shift and the inverse FFT come along for the cost of some latency until the pipeline is filled, a few hundred nanoseconds. That translates into getting a speed up for the phase shift and the inverse FFT for free. The overall throughput improvement can then be estimated as follows:

4.5 + 4.5 + [(25/35)*4.5] = 12.2x (the overall speed up for the pieces we put in hardware).

In this calculation, [(25/35) *4.5] is an estimate of the performance of the phase shift. It is based on the fact that the phase shift uses similar math to the FFT and the 25/35 is the proportion of the time taken on the CPU for the FFT.

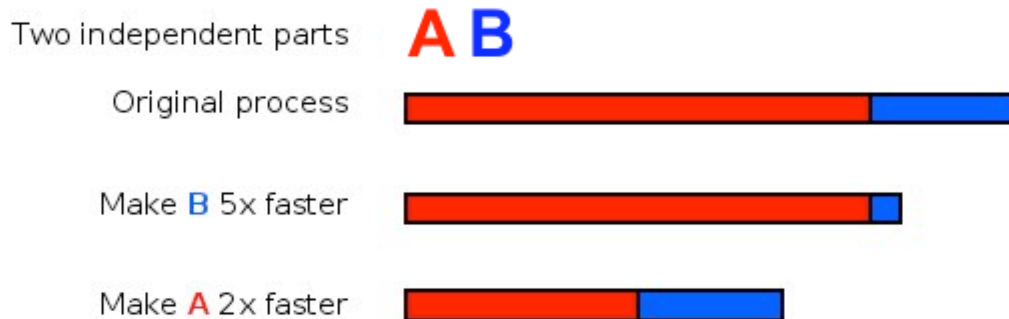
Using Amdahl's Law P = the percent that can be parallelized and S is the speedup of that portion (See Amdahl’s Law – An Overview for a quick summary of the law). Using this formula we find

$$\frac{1}{(1-P)+\frac{P}{S}}$$

= 1/[(1-0.95) + (0.95/12.2)] = 7.8x application speed up.

Today, there are few commercially available tools to automatically do the software parallelization. And those that are available are still in their infancy. Tools will have to examine issues such as coarse-grain vs. fine-grain parallelism, recursion, data dependencies, data flow, and various performance aspects such as load balancing between the host system CPU and the accelerator (bus hogging, memory access bandwidth, data transfer bottlenecks, etc.).

Amdahl’s Law – a quick overview



Explanation
Assume that a task has two independent parts, A and B. B takes roughly 25% of the time of the whole computation. By working very hard, one may be able to make this part 5 times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part A be twice as fast. This will make the computation much faster than by optimizing part B, even though B got a bigger speed-up, (5x versus 2x).
(explanation taken from caption in wikipedia:Amdahl's law)

About the Author:
Steve Casselman, CTO and Founder

Steve is co-founder of DRC and has been instrumental in shaping the reconfigurable computing industry. He holds 5 issued patents and received the first SBIR Technology of the Year Award for his work with the Naval Surface Warfare Center. He received his Bachelor of Art in Mathematics from U.C.L.A.