



[www.drccomputer.com](http://www.drccomputer.com)

# Software to Hardware Parallelization

To accelerate algorithms on multi-core systems, you must first identify the code within the application that can be parallelized, then figure out how to parallelize it.

Steve Casselman, CTO and cofounder, DRC Computer

May 20, 2008

---

Most of the software written today involves instructions executed in sequence. To speed up execution, programmers have typically pushed hardware designers to build processors with increasingly higher clock rates, resulting in heavily pipelined processors that operate at clock rates of 3 GHz and more. To get the most out of every clock cycle, these processors resort to tricks such as large caches and out-of-order execution.

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

Hardware designers have turned to the latest generation of 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) for 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 1000 times over general-purpose processors. Also, any algorithm that needs billions of integer or floating-point operations per second (image and audio processing, seismic exploration, bioinformatics, and the like) can be accelerated by an FPGA-based coprocessor.

To accelerate an algorithm, you must first identify the code within the application that can be parallelized, then figure out how to parallelize that code so it can be executed on an array of computational elements configured in an FPGA or ASIC. But the question is where to start. The logical starting point is to first profile the code to find its computationally intensive portions, then find ways to isolate the code so that it does not have many data dependencies. Once the code has been isolated, you need to find ways to optimize it so that it can be executed on the resources available on the coprocessor. It is difficult to optimize the code to fit 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, make it easier for you to define an optimized architecture on a case-by-case basis.

The next step to accelerating an algorithm is to examine the communication channels between 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, ensure that the hardware can keep up with the pace at which the software feeds data to the hardware. Finding the right balance in the hardware/software partitioning between system inputs and outputs is critical to smooth operation. If it takes longer to transmit data to the accelerator than it takes the CPU to calculate the answer, make sure that the hardware design uses all memory accesses most efficiently. You may even consider adding 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. The data then comes out of the memory again and the algorithm executes the phase shift computations and places the data back into memory one more time. Finally, the inverse FFT retrieves the data, executes the algorithm, and then places the final result back in memory. Main memory accesses are expensive, so reusing data can dramatically improve performance by eliminating multiple store and access operations.

When performing numerical calculations, you should track the precision of the variables that really need to be implemented—you may not always need the full precision of single- or double-precision floating-point data types for every variable. A fixed-point multiplier and arithmetic unit combination 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 faster than floating-point computations.

To parallelize the algorithms, you must first perform a data-flow, which helps you understand how data moves between the different logic and computational elements. Next, execute a latency analysis to 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, you use PCI Express or HyperTransport to provide high-bandwidth, low-latency communication channels.

Last, 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. Consider different types of math for computations aside from basic integer and

floating-point operations. Logarithmic computations or math based on residue number systems, for example, could execute faster than standard integer or floating-point-based code.

In seismic processing software, wave migration algorithms often incorporate FFT as part of the solution. This algorithmic technique (the FFT approach) usually takes the following sequence: Apply the FFT, phase shift the data, and then apply the inverse FFT to obtain the next location of the traveling wave front. When this is computed on a processor via software, data is taken out of memory to compute the FFT, then the result is put back into memory. The resulting data is taken out of memory again to do the phase shift and the new result is 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 that can slow down the execution. Using hardware to accelerate the algorithm, begin by taking the data out of memory and feeding it to a pipeline that executes the FFT and feeds the result into the phase shifter. The phase shifter then 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 because 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 5:1 on floating-point code, but can deliver a 10-100 time improvement on integer-based code.

Next, see if the code fits into the hardware. Take a look around the Web to find a variety of floating-point and integer FFT algorithms. Floating-point versions are larger and slower, but deliver more precise results. One example, a hybrid approach, can compute a 2Kpoint FFT at a rate of 400 Msamples/s ([www.andraka.com/V4\\_FP\\_fft.htm](http://www.andraka.com/V4_FP_fft.htm)). This is 4.5 times faster than a 2.2-GHz dual-core Opteron ([www.fftw.org/speed/opteron-2.2GHz-64bit](http://www.fftw.org/speed/opteron-2.2GHz-64bit)). Three such FFT engines can fit in a Xilinx SX55. A CORDIC algorithm can be used to do the phase shift, because implementing such an algorithm does not consume many of the FPGA's logic elements—less than 1000 look-up tables. Moreover, the computations can be pipelined to improve throughput.

While you may be impressed with speeds 4.5 times faster, you might still think that you can buy a 4-way Opteron-based system for the same price as an FPGA-based coprocessor with a dual-core Opteron to achieve the same performance gains. But because the functional units will be pipelined in an FPGA-based coprocessor, the phase shift and the inverse FFT come with the cost of some latency until the pipeline is filled—a few

hundred nanoseconds. As a result, you achieve a speedup 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 speedup for
the pieces we put in hardware)
```

The calculation  $[(25/35)*4.5]$  is an estimate of the performance of the phase shift and is based on the phase shift using 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 equals the percent that can be parallelized and S is the speedup of that portion. Using this formula you find that:

```
1/(1-P)+(P/S) = 1/[(1-0.95) + (0.95/12.2)] = 7.8x application speedup
```

The Advanced Encryption Standard (AES) algorithm is another CPU-intensive piece of code. The main pseudocode is:

```
case 128: ROUNDS = 10; break;
case 192: ROUNDS = 12; break;
case 256: ROUNDS = 14; break;
*/
    KeyAddition(data, round_key[0]);

/* at most ROUNDS-1 ordinary rounds */

    for(r = 1; (r <= ROUNDS) ; r++) {

        Substitution(data);

        ShiftRow(data);

        MixColumn(data);

        KeyAddition(data, round_key[r]);

    }

/* do the last, special, round: */

    Substitution(data);

    ShiftRow(data);

    KeyAddition(data, round_key[ROUNDS]);

}

done;
```

The AES algorithm can be broken down into four subroutines: *Substitution*, *ShiftRow*, *MixColumn*, and *KeyAddition*. In the aforementioned sample, one block of 16 bytes as a 4×4-byte matrix is run through the entire algorithm. In a pipelined version, when one block of data is encrypted with the first subroutine, it moves to the second subroutine and a new block of data is initiated in the first routine—coarse-grain parallelism at the function level. In addition, each subroutine may run in parallel (fine-grain parallelism). As an example, let's look at the *ShiftRow* routine.

The *ShiftRow* routine modifies each row of the state matrix. The top row is not changed; the next row is rotated left one position, the following row two positions, and the bottom row three positions. A processor must modify each row, then go to the next row. In an FPGA, all rows can be changed in one clock.

Today, there are few commercially available tools that automatically do software parallelization, and those that exist are still in their infancy. The next generation of tools will have to address issues such as coarse-grain versus fine-grain parallelism, recursion, and data dependencies and flow, and various performance issues such as load balancing between the host system CPU and the accelerator (bus hogging, memory access bandwidth, data transfer bottlenecks, and the like).

