Massively parallel processing arrays (MPPAs) for embedded HD video and imaging (Part 2)

Laurent Bonetto
Technical Marketing Engineer,
Ambric, Inc. - July 18, 2008

In the previous article, "Massively Parallel Processing Arrays (MPPAs) for Embedded HD Video and Imaging - Part 1", we provided an overview of the various types of architectures used to meet the needs of high-performance embedded applications. This analysis brought us to discuss in more detail Massively Parallel Processing Arrays (MPPAs), a type of programmable architecture with computational power similar to FPGAs and superior to high-end DSPs, and a programming model that is appealing to software developers. In that context, we introduced the Ambric Am2045 MPPA architecture and its Structural Object Programming Model (SOPM).

Target application example: JPEG codec
This second article illustrates how to apply this programming model to successfully implement embedded applications by using a JPEG codec and discussing its implementation on the Am2045.

Relevance of JPEG
JPEG is a standard for image compression. Or rather, JPEG is the standard for image compression today. While the video compression space is cluttered with a sea of codecs like MJPEG, MPEG-2, MPEG-4, H.264, AVC Pro, Xvid, WMV -- just to name a few -- JPEG seems to dominate the image encoding space.

The JPEG compression format is also found in the video space in the MJPEG video codec, which is essentially a sequence of JPEG images put together.

Finally, almost all video compression standards share many functional blocks with JPEG. Most video codecs operate on small blocks of pixels (8x8 or similar sizes) on which they perform similar operations such as transformation into the frequency domain (DCT or similar algorithms), quantization, run-length encoding, and so on.

JPEG is a representative and realistic example of a high-performance embedded application, and JPEG is simpler to describe than many other high-end applications, making it practical for the context of this paper. The process for implementing JPEG on the Am2045 is identical to what is required for developing other high-end applications, such as the existing Am2045 GT Video Reference Platform that can be used for H.264, MPEG2, JPEG2000 and many other powerful codecs.
### Algorithm overview

In the remainder of this article, we use the term JPEG to refer to the most common JPEG mode: baseline JPEG.

JPEG is a lossy image compression codec. The encoder transforms an image into a compressed bit stream using the following operations:

- Each image is converted to a chroma-luma color space where the chroma pixels can be downloaded to provide a first opportunity for compression.
- The image is divided into 8x8 blocks of luminance (Y) and chrominance pixels, which are transformed into the frequency domain using a Discrete Cosine Transform (DCT). This transformation decorrelates the image, concentrating most of the energy into a few (low-frequency) coefficients.
- Each DCT-transformed block is then multiplied by a quantization matrix, which provides a "knob" to adjust the level of compression: Increasing the quantization deteriorates the image quality but increases the compression ratio.
- Each quantized coefficient is then read in a zigzag order, a pattern that orders the coefficients from low to high frequencies. This introduces large groups of consecutive zeros toward the end of each block.
- The coefficients in zigzag order are then run-length encoded, a process that encodes each non-zero coefficient as a pair where the first number represents the number of preceding zeros and the second number represents the value of the non-zero coefficient. This step allows us to compress long runs of zeros.
- Finally, the run-length codes are entropy coded using Huffman tables. These tables allocate fewer bits to the most common codes, thus reducing the average size needed to represent each code.
- The resulting string of Huffman code is packed into a JPEG file following a header containing the information needed to decode the file, such as the quantization and Huffman tables used for encoding each color component.

An additional compression step consists of encoding the first coefficient of each block (the DC coefficient) as a subtraction from the DC coefficient of the preceding block, thus taking advantage of the similarities between neighboring blocks.

The JPEG decoder simply applies the transformations described above in reverse order: the decoder first extracts and decodes the Huffman codes from the packed bit stream, then rearranges the data from a zig-zag order to a block order, performs an inverse quantization, and finally an inverse DCT (IDCT).

### Next: Design development

**Design development**

Most software developers undertaking a complex embedded design development break that process into three major steps:

- Creating a functionally correct design as quickly as possible
- Optimizing the design to meet the system-level objectives
- Executing the design on hardware

This article will now discuss each of these steps in detail and explain how each step compares to an MPPA as compared to other conventional architectures such as FPGAs and high-end DSPs. We also will illustrate this development process by discussing the implementation of the JPEG codec on the...
Ambric Am2045 MPPA architecture using Ambric aDesigner software development tools. This approach demonstrates how an MPPA supported by proper software development tools offers a straightforward path to creating optimized designs.

Naturally, software developers are not restricted to follow the three-step approach discussed in this article and aDesigner is designed to offer that flexibility. For example, a software developer may need to develop prototype blocks and then test them directly in hardware prior to optimizing them. Given that the Ambric aDesigner tool suite takes only a few seconds to about a minute to place and route designs, developers can follow this approach successfully.

**Functionally correct design**
A common task for software developers is to produce a reference design that will run, if not on the target chip, at least in the development environment of that chip. That design isn't expected to run fast, but it needs to produce correct output so that it can be used as a starting point for progressively creating a fully optimized implementation.

Usually, the design being produced is abstracted from most features of the target architecture, which allows the design to be less complex and more general.

aDesigner provides a programming environment that simulates Java code without compiling or running the code on the actual chip. This environment is perfectly suited for developing generic code since no architecture limitations are enforced -- an arbitrary number of processors and channels can be used and the amount of on-chip memory available is unlimited.

High-level algorithm descriptions usually come in the form of detailed specifications and their associated reference code usually written in MATLAB or C. In the case of JPEG, our starting point was the ITU T.81 specifications and one of the most commonly used, freely available, and reasonably optimized C reference source codes for JPEG: the IJG JPEG library (version 6B).

The IJG JPEG library is a good representative of the kind of reference code that embedded software developers start from: it contains a large amount of code not needed for the target embedded application such as, in our case, code supporting multiple JPEG modes other than JPEG baseline. In this kind of situation, the software developer is left with two approaches. He or she can become familiar with the entire reference code and trimming down what isn't needed, or write new reference code from scratch and borrow code sections from the reference code when practical. We chose the latter approach. Both approaches usually require a comparable amount of effort.

**Next: Test vectors, allocating processors, the cycle budget**

**Test vectors**
A step that can be time consuming when developing a design from scratch is creating test vectors to validate progressively the functionality of the blocks as they are created. We solved that challenge by developing the JPEG encoder and decoder in parallel so that each encoding and decoding block could be used to test the other without any need to create reference output data.
Allocating processors
The resulting reference JPEG implementation is a very natural decomposition of the algorithm into a small number of processors each of which independently runs a section of the JPEG algorithm roughly corresponding to one of the blocks presented earlier in this article in the Algorithm Overview. The process of allocating processors to tasks flows naturally from the desire to keep the implementation simple to write.

For example, one processor successively runs the zig-zag encoding block followed by the quantization block by simply reading the input data for the quantization in a zig-zag order. Another processor runs the run-length encoding block and Huffman encoding blocks together because devising a way for these two blocks to communicate efficiently while running independently on two processors would be cumbersome.

Following the same principle of easing the work of the software developer, other blocks can instead be divided across multiple processors. This was the case with the byte-stuffing operation required by the JPEG standard: byte-aligned 0xFF symbols must be followed by a zero byte. Writing code that successively packs bit strings of arbitrary sizes into bytes, performs byte-stuffing, and then packs the byte stream into 32-bit words is difficult and error-prone. It also results in code that is excessively long, is difficult to follow and to optimize in assembly, and has low throughput.

On a conventional architecture, such a sequential approach is the only approach available because dividing the work into smaller chunks -- for example by using multiple threads or multiple function calls -- introduces too much overhead. On the other hand, with an MPPA, dividing the work makes it simpler and more importantly doesn't add cycle overhead. We used one processor to pack the bit strings into 32-bit words and used another processor to read these words one byte at a time and generate a stream of bytes that has been byte-stuffed. The resulting code running on each processor is both simpler and more efficient.

In summary, we can use the hundreds of processors available on the MPPA to our advantage whenever it makes the coding simpler. And since all processors stall automatically when reading from an empty channel or writing to a full channel, no code is required for inter-processor synchronization.

Creating the actual design
We will now discuss the amount of work involved in creating the actual design. Creating a design from scratch using Java requires a level of effort comparable to creating the same design in C and much lower than what is required for using RTL. Most blocks were simple enough to be written from scratch with only two exceptions:

- DCT: A good DCT implementation takes months to develop, and it makes sense to leverage existing optimized code. We reused the DCT code found in the IJG C reference code. How much effort was required to convert that C code into Java code? Zip! To be more precise, we had to replace five invocations of the macro MULTIPLY that implemented a fixed-point multiply by an actual multiplication instruction. In other words, all of the C code used in the IJG DCT compiles readily as Java code.
- Initialization routines for creating the Huffman lookup tables: The Huffman tables encoded in the JPEG header are in a compact form. Any efficient JPEG implementation requires these to be
expanded. The process for expanding these tables doesn't need to run fast but isn't trivial to write. Again we reused the code provided with the reference standard. Effort to convert that C code into Java code? None. All of the C code used in the IJG Huffman initialization routines compiles readily as Java code. The only changes consisted of renaming variables to follow a different coding convention.

These two examples illustrate the point made in the first article that embedded code written in Java and C is most often identical.

Next: Optimizing the design

Optimizing the design

Optimizing an application to meet its requirements is the most common and time-consuming task that embedded software developers face.

The first step is common to all architectures: translate the application requirements into a cycle budget. That is, determine how many cycles we have available to consume an input word or produce an output word.

After determining the cycle budget, the software developer must devise a plan to meet that cycle budget.

- With conventional DSPs, the approach consists of sharing that cycle budget across all functions in the target application. The DSP developer profiles the entire application, identifies the most time-consuming functions, and proceeds to optimize them until the entire application runs fast enough.
- With FPGAs and ASICs, the approach consists of developing and validating faster hardware, with more parallelism, function-specific operations, etc.

MPPAs combine the benefits of both approaches:

- Like an FPGA, but unlike a conventional DSP, an MPPA gives the software developer the option of trading area for speed. This allows software developers to accelerate blocks of arbitrary complexity with no significant rework of the code. Time-consuming blocks can be parallelized onto multiple processors using functional or data parallelism. Functional parallelism consists of running different parts of an algorithm on successive processors. Data parallelism consists of running the same algorithm on independent blocks of data using different processors.
- Like a conventional DSP, but unlike an FPGA, the MPPA enables the software developer to optimize the code running on a single processor in software without the cumbersome process of developing and verifying RTL code. Code that requires a moderate increase in speed usually remains written in a high-level language, relying occasionally on intrinsics to use processor instructions when they provide a significant code speedup. Code that requires more speedup is usually written in assembly. This optimization process is simpler on an MPPA than on a high-end DSP for multiple reasons:
  - The functional code to be optimized on an MPPA is usually simpler to start with, as was explained in the previous section.
  - Each individual processor on an MPPA is simple. As a result, the software developer doesn't have to rely heavily on optimization tricks such as code pipelining, loop unrolling, or algorithm restructuring to keep multiple execution units busy at every cycle.
  - With MPPAs, when a chain of processors executes a section of a design, only the processors that are the bottlenecks and do not meet their cycle budget need to be fully optimized. With high-end
DSPs, all functions that contribute noticeably to the total cycle count must be heavily optimized. Software developers who have spent weeks hand-tuning assembly code to save a few extra cycles in a single function will appreciate the appeal of the MPPA optimization process.

The quality of the development tools also has a key impact during the optimization phase because this is usually where most of the time is spent. The aDesigner tool set offers all conventional debugging features that are standard in the industry today. In addition, it facilitates the debugging and validation of any code being developed by letting the user insert taps on the input and output ports of any processors. This technique allows developers to collect reference input and output data to debug any single processor or quickly identify the source of a bug across a design of arbitrary complexity.

In summary, software developers can optimize MPPAs in behavioral programming languages, which are much more practical than the RTL language that is practically unavoidable for FPGA and ASIC users. At the same time, simpler target processors combined with less stringent requirements for each processor's code makes fully optimized assembly code less difficult to write and less often required than with conventional high-end DSPs.

**Next: Determining and meeting the cycle budget**

**Determining the cycle budget**

The first optimization step is to determine a cycle budget. Our primary goal for this project was to encode at average quality sixty 640x480 images a second, which, assuming a 300 MHz processor clock speed, translates into a cycle budget of 5.4 cycles per incoming color component. Since JPEG is a compression codec, the data rate decreases as we progress in the processing chain and the cycle budgets increase accordingly as illustrated in Table 1 below.

<table>
<thead>
<tr>
<th>Cycle budget</th>
<th>Per input byte (R, G, or B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>~9 cycles</td>
<td>Per Huffman code produced</td>
</tr>
<tr>
<td>~90 cycles</td>
<td>Per 32-bit output word</td>
</tr>
</tbody>
</table>

*Table 1: Cycle budget for the encoder target*

Note that the first number (cycle budget per input byte) is independent of the quantization and the image being encoded. The following two numbers vary depending on the image being encoded and the quality settings being selected. As a result, these two numbers are padded to guarantee that the encoder meets the target throughput when encoding any image at a similar quantization level.

**Meeting the cycle budget**

The reference design and the fully-optimized design both run in the same aDesigner environment, which allows optimization to be done progressively one step at a time.
The aDesigner's profiling tools enable us to measure how many cycles each processor takes to produce an output sample and compare these numbers to the cycle budgets above. Not surprisingly, complex blocks like the DCT that are at the beginning of the processing chain and perform a lot of computation on each pixel require the most optimization and as such must be written in assembly. Blocks like the bit string packing that come at the end of the processing chain with a large cycle budget per output sample require no modification and can remain in Java.

On MPPAs, as with conventional DSPs, some functional blocks lend themselves better than others to the parallelism available at the assembly instruction level. For example, blocks like color conversion, DCT, or quantization can take full advantage of instructions operating on half words while some portions of blocks like run-length and Huffman encoding must remain serial (i.e., processed only one sample at a time). As a result, after first-order optimization of blocks that is required in assembly or in Java was completed, we compared the cycles spent by each function to the cycle budgets described in Table 1.

We found that only the run-length and Huffman encode block exceeded its cycle budget significantly. It exceeded its cycle budget by about 30%.

With a conventional DSP, a software developer not able to meet his or her target requirements is confronted with two choices: further optimize the most time-consuming functions that have already been optimized to a first order (i.e. save a few more cycles on functions that have a big impact on performance), or optimize more of the less time-critical functions (i.e. save a larger number of cycles on functions that have a small impact on performance). Each is too time consuming to be desirable -- each small additional increase in performance comes at the expense of a large amount of work.

With an MPPA like the Am2045, the software developer can use a simpler third approach -- apply data parallelism across more processors. We chose this approach for improving the run-length and Huffman encode step. Two processors are assigned to run the same run-length and Huffman program in parallel. This general approach speeds up any code of arbitrary complexity with only trivial changes to the code. In our case, this technique allowed the run-length and Huffman encode block to go from exceeding its cycle budget by 30% to outperforming by about 25%.

**Next: Resulting implementation**

**Resulting implementation**
The resulting design fits in a few SR and SRD processors as shown in Figure 1. The design relies on a combination of Java and assembly code. Longer non-time critical code such as the initialization routines expanding the Huffman tables or those creating the JPEG file header is written in Java and shorter time-critical code is written in assembly.
Due to the simplicity of the MPPA processors, the assembly code is straightforward: the most complex block (run-length and Huffman encode) consumes about 100 assembly instructions. The second most complex blocks (horizontal and vertical DCT) run practically the same code: around 60 assembly instructions. Other assembly blocks average about 20 assembly instructions each. Normal programming practice was used to write the assembly code and no attempt was made to get unduly creative. Nonetheless, as can be seen, these numbers are much lower than what would be seen on a typical high-end DSP.

In summary, we wrote approximately 300 lines of assembly code and used only about two brics out of 45 available to run the entire JPEG encoder and meet our application requirements. Most resources (except the SDRAM used to buffer macro blocks) are contained within these brics with complete encapsulation. This leaves ample resources for other applications to execute in parallel on the chip without any risk of affecting the functionality of the JPEG encoder or deteriorating its performance. The same is not true with conventional DSPs, which rely on time sharing the CPU in order to run multiple applications all at the same time, an approach that adds resource switching overhead and, more importantly, affects the contents of cache, thus resulting in poorer performance when applications are combined.

The SDRAM bandwidth utilization is in general well balanced with the on-chip resource utilization (processors and local memory). This is the case with the JPEG implementation, which uses about 4% SDRAM bandwidth against the 5% utilization of on-chip resources discussed above.

In terms of processing throughput, each processor meets its cycle budget, and therefore the implementation exceeds the requirements. The bottleneck happens to be the processor packing the Huffman codes into a bit string: it consumes an average of 7.5 cycles per Huffman code being consumed. Therefore the overall JPEG encoder achieves a throughput of approximately 72 frames per second against the target of 60 frames per second.

Next: Executing the design on the chip

Executing the design on the chip

Most of the design development and optimization occurs in a simulation environment, which is typically the preferred development environment for software developers. The next step for the
developer is to take a design that produces correct output on a simulator and run it on the chip to ensure consistent functionality and meet design speed constraints. This step requires the following considerations:

- **Fitting code into on-chip memories**
  Almost all code developed by the process described in the above sections is already broken down into pieces small enough to fit into the local memory of each processor. Code that does not fit into the local memory of one processor can either be split onto several processors or further optimized for space by using more function calls, when this is an option.

- **Fitting local data arrays into on-chip memories**
  The Am2045 is a non-cached architecture, which means that the software developer is in full control of which data arrays go into on-chip memory and which remain in external memory. This architecture results in deterministic performance, a key advantage over cached architectures. aDesigner automatically allocates up to four memory banks (for a total of 8 KB) to each data array or group of data arrays created by the software developer. As a result, most arrays fit readily into on-chip memory. Again, the most common exception is for processors running a large amount of non-time-critical code where a large number of small arrays are typically present. When combined, these arrays may end up taking more space than is available in an RU's memory. Common solutions include reusing the same arrays for temporary calculations, performing in-place computations in which the output buffer replaces the input buffer, or using more than one processor to run the initialization code (because each processor can access a different RU's memory banks).

- **Placing and routing the design**
  Due to the large granularity of the design, the place and route task on an MPPA is considerably simpler than it is on an FPGA or ASIC. aDesigner handles the place and route by deciding which processors and local memories to use to run the application, and how to connect processors and memories according to the software developer specifications. This step takes less than a minute to complete, even for a large design that uses most of the resources on the chip.

- **Running the design at the expected speed**
  Running an application on the chip allows the developer to obtain a full and accurate characterization of processor activity that takes into account system level issues such as SDRAM latencies. aDesigner captures a number of types of profiling data measured on hardware at runtime, i.e. processor % utilization, channel activity, instruction profiling, etc. By displaying profiling data on a time axis using an intuitive graphical color-coded display, aDesigner allows the software developer to identify situations that may cause an application to underperform such as a processor on a time-critical path that remains inactive as a result of an insufficient buffering of data.

**Next: Running the design on hardware**

**Running the design on hardware**

The Am2045 software developer uses the same aDesigner integrated development environment to target the chip or use the simulator. aDesigner allowed the entire code and data arrays to fit readily into local memory, including the initialization routines. For example, aDesigner automatically allocated one additional bank to store the larger code that handled the block dispatch and initialization of quantization and Huffman tables as it didn't entirely fit into the processor's local instruction memory. The design automatically placed and routed in less than 5 seconds.

Running the design on hardware identified one object that had been left in Java and was too slow "the byte stuffing object. It included a time-critical loop with multiple tests that were better handled in assembly than by the compiler. After that object was optimized, aDesigner's profiling tool, which non-intrusively samples processor activity at run time, confirmed that the processor expected to be
in the critical path (the object performing the bit packing) was kept busy 100% of the time. This confirmed the analysis that had been made in the simulator environment. As a final validation step, we confirmed that we were able to encode more than 60 640x480 frames in less than a second using different quantization tables leading to the desired image quality.

Programming with MPPAs: Conclusion
The JPEG implementation on the Ambric Am2045 MPPA architecture shows a programmable implementation that can easily be scaled to achieve levels of performance higher than any high-end DSP and comparable to those achieved by many FPGAs.

The following summarizes the key common points or differences between creating a design on an MPPA versus using other programmable architectures targeted at high-end applications:

- **Partitioning**: Embedded designs running on conventional DSPs are usually divided into multiple functions. With MPPAs, these functions map logically onto separate processors without the overhead of function calls.
- **Behavioral language**: The availability of a behavioral software language to develop a reference implementation that closely models the final algorithm implementation is a key advantage compared to FPGAs and ASICs, which rely on the RTL language to produce a realistic model of any implementation.
- **Code safety**: Unlike other parallel architectures, MPPAs guarantee that each processor has its own local memory that cannot be affected by other processors running on the chip. All processor communication is handled safely and efficiently through self-synchronizing channels. Also, bugs inherent to C such as pointer errors disrupting unrelated code are simply not possible when the underlying programming language is Java, as is the case with the Am2045.
- **Debugging**: The process of debugging applications on an MPPA is similar to what software developers experience with conventional DSPs in that each function can be implemented and tested one at a time. When a bug occurs in a design, port taps allow a software developer to analyze quickly the streams going in and out of any processor in a design to identify quickly the source of the bug.
- **Optimization**: Optimization is easier on MPPAs than on any other conventional architecture due to the simplicity of each processor. This advantage is critical as software developers spend a lot of time writing code in assembly to meet performance requirements. In addition, the ability to use data or functional parallelism provides software developers with additional optimization options.

Editor's Note: This is the second of two articles discussing how Massively Parallel Processing Arrays (MPPAs) can be used to accelerate high-performance embedded system applications. "Massively Parallel Processing Arrays (MPPAs) for Embedded HD Video and Imaging - Part 1" discusses the requirements of high-performance applications and how MPPAs compare with other architectures.

About the author
Laurent Bonetto is technical marketing engineer at Ambric. His working focus at Ambric is on the aDesigner comprehensive development tool suite and the Am2000 family of chips and boards. Mr. Bonetto is also involved in defining architecture instruction sets and developing application firmware at Ambric. He has expertise in a wide range of processor architectures. Previously, Mr. Bonetto worked at Cradle and, prior to that, at Berkeley Design Technology Inc. (BDTI) where he was benchmarking and analysis program manager. Mr. Bonetto received his ME degree from the Georgia Institute of Technology. He also holds undergraduate degrees in mathematics and engineering from the Electrical and Computer Engineering School, Supelec, France. He can be
reached via info@ambric.com.