Comparing Fine-Grained Performance on the Ambric MPPA against an FPGA

Brad Hutchings, Brent Nelson, Stephen West, Reed Curtis
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering
Brigham Young University
Provo, UT 84602 *

Abstract

A simple image-processing application is implemented on the Ambric MPPA and an FPGA, using a similar implementation for both devices. FPGAs perform extremely well on this kind of application and provide a good benchmark for comparison. The Ambric implementation starts out with a naive implementation and proceeds through several design optimizations until it reaches a maximum frame rate of 164 FPS (512 x 512 images) which turns out to be approximately 7x slower than the FPGA. The final Ambric implementation uses only 18 of 336 available processors, achieves more than sufficient performance for real-time embedded applications, and has excess processors to use for implementing additional algorithms. After introducing the image processing application and its implementation on both devices, the paper compares and contrasts the intrinsic, general characteristics of Ambric MPPA and FPGA devices.

1 Introduction

For over 2 decades, FPGAs have been used to accelerate applications in a wide variety of areas. At first, there were really only two alternatives to the FPGA: programmable uniprocessors such as microprocessors or DSPs, and Application-Specific Integrated Circuits (ASIC). Most published application studies have typically compared FPGA performance against those 2 yardsticks (and the occasional supercomputer). However, relatively inexpensive multiprocessor devices such as GPUs and Massively Parallel Processor Arrays (MPPA) have arrived and are achieving similar levels of performance in many cases [2, 3]. Moreover, these devices are proving to be easier to program than FPGAs [4].

*This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. 0801876.

Recently, an MPPA came to market from Ambric, Inc. Consisting of 336 processors that are programmed in Java, the Ambric device is a highly parallel device targeted at embedded applications. To better understand the relative strengths and weaknesses of the Ambric device, this case study selected a simple image-processing algorithm — Sobel edge detection —- that is known to perform well on FPGAs, implemented it on both an FPGA and an Ambric MPPA and compared the two implementations. The application was implemented in a way to exploit the fine-grained architecture of the FPGA so Ambric's performance on fine-grained applications could be studied.

This effort sought to answer the following specific questions:

- *Performance-wise, how close is the MPPA to the FPGA for fined-grained computation?*
- *What features of the MPPA limit performance for fine-grained applications?*

The rest of this paper is organized as follows. First, the Ambric device architecture and programming model is introduced followed by a description of the Sobel edge-detection algorithm. The next section‘, discusses the VHDL-FPGA and Ambric implementations of the Sobel edge-detection algorithm. Finally, the next section contrasts, in a general way, the relative strengths and weaknesses of the FPGA and MPPA.

1.1 Ambric and Its Programming Model

The device used in this work is the Ambric AM2045 Massively Parallel Processor Array (MPPA). The Ambric MPPA contains 336 32-bit processors and 4.6 Mbits of SRAM. The entire array is synchronous and operates at 300 MHz. It is a standard-cell ASIC containing 117 million transistors and was fabricated at 130 nm [2].

The AM2045 is internally organized into a 5×9 array of *bric* modules. Figure 1 shows one bric and its neighboring

978-1-4244-3892-1/09/$25.00 ©2009 IEEE

brics[1]. Each bric contains two kinds of 32-bit CPUs. SRD processors contain 3 ALUs and provide math-intensive instructions to support DSP operations. Each SRD processor contains a dedicated 256-word RAM for instructions and data. This memory can be augmented though direct connections to bric memory objects. SR processors are lighter weight and contain only 1 ALU and are often used for tasks such as address generation. They contain a dedicated 128-word memory for programs and data but do not have direct connections to memory objects. Each of the two memory objects (RU) in a bric is organized as 4 independent RAM banks.

The intra-bric communication paths constitute the level 1 communications in the chip. Level 2 communication channels provide direct connections to neighboring brics as shown in the figure. These are non-shared channels and provide high bandwidth. The level 3 interconnect is for long-distance communications and consists of a chip-wide 2D circuit-switched interconnect of channels. These longer channels share physical resources and thus provide less bandwidth than nearest neighbor channels.

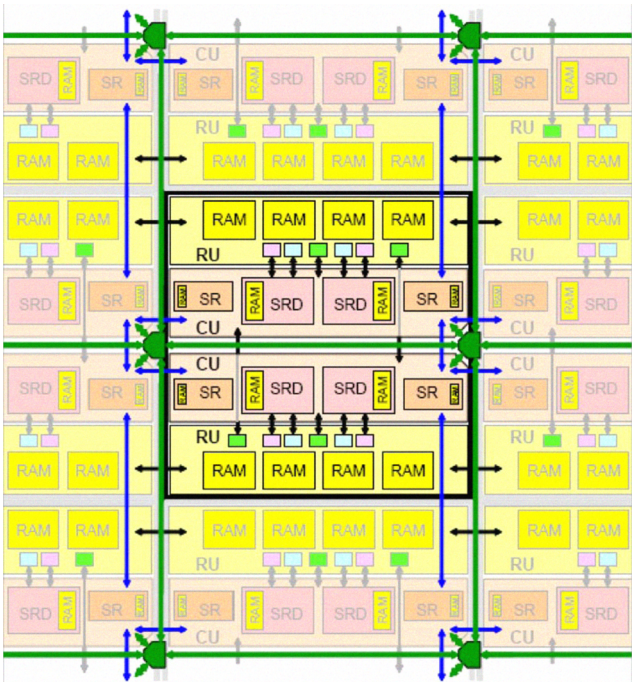


Figure 1. Ambric *bric* Organization

As Figure 2 shows, the AM2045 chip also contains a variety of external interfaces: two 32-bit DDR2-400 SDRAM interfaces, a 4-lane PCI Express interface for chip configuration and data transport, a serial flash interface, a JTAG interface, and 128 1-bit general-purpose parallel I/O ports.

[1]Figure 1 and Figure 2 used by permission of Ambric.

Additional details of the AM2045 can be found in [1, 2].

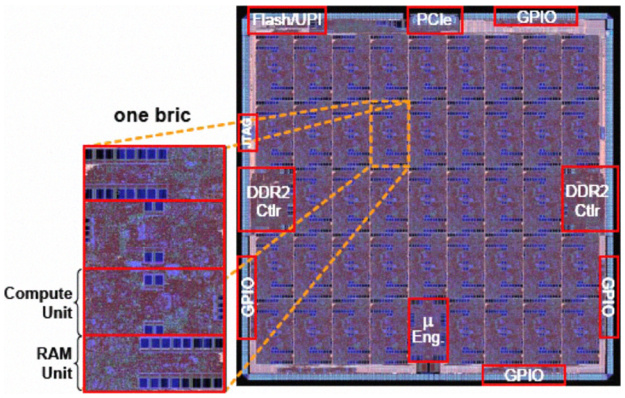


Figure 2. Ambric Chip Organization

There are two basic primitive objects in the Ambric programming model: processors and memories. The memory objects in each bric can be used in four different ways: (1) as data storage for SRD processors (FIFO or random access), (2) as instruction storage for SRD processors (FIFO or random access), (3) to implement FIFOs between processors, and (4) as random-access memory accessible over the MPPA's network. Multiple memory objects can be combined to create deeper FIFOs.

Processors and memory objects communicate over channels that are word-wide, point-to-point and strictly ordered. Channels behave like synchronous FIFOs and are blocking. Channels are self-synchronizing, using a tagged approach similar to that found in data-flow machines. Reads from an empty channel cause a processor stall as do writes to full channels. Self-synchronizing channels are key to the Ambric programming approach. They allow individual processor and memory objects to operate independently at their own speeds, synchronizing as they receive and transmit data on their respective channels.

Programmers develop applications on the Ambric MPPA by writing small Java programs, one per processor. The programmer also provides a "structural" description of the application that assigns programs to processors and defines how processors and memories are connected together by channels. Thus, creating an application for Ambric feels, in many ways, similar to hardware design. The Ambric compilation process produces an image which is downloaded into the chip, not unlike an FPGA bitstream, which configures each processor and the interconnections between them.

1.2 The Sobel Image Processing Kernel

The computation used in this work is the Sobel operator, a 3×3 image convolution kernel commonly used for edge

detection. It calculates the gradient of the image intensity at each point. Separate kernels exist for computing the gradient in the x and y directions. Mathematically, the x gradient image G_x is computed by

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \times A$$

where A is the input image. The y gradient is computed similarly but with a transposed kernel.

In custom hardware, image convolutions such as this are often computed using a general structure similar to that shown in Figure 3. The raw image data is fed in from the left in row-wise order. Each delay line buffers up one row of the image, allowing a sliding 3×3 neighborhood of image pixels to be formed in the flip flops to the right. The actual convolution, consisting of multiplies and adds, is then performed on the neighborhood to produce one result pixel for each such neighborhood. The *Convolve* block computation may be done sequentially using a single multiplier and accumulator or in a parallel pipelined fashion, depending on the processing rates required by the application.

A typical application of Sobel is to compute both G_x and G_y convolutions for a given neighborhood and then produce a final image as $G = |G_x| + |G_y|$. This is easily done by augmenting the arithmetic logic in the *Convolve* block in Figure 3 to concurrently produce both gradient results and combine them into G.

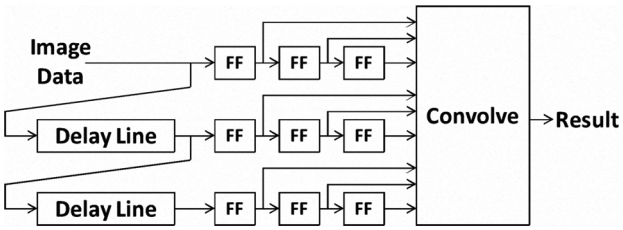


Figure 3. Image Convolution Hardware

2 Implementing Sobel on the FPGA

The FPGA implementation of Sobel was done on a Virtex 4 FX12 part (xc4vfx12-10ff668) and closely follows that shown in Figure 3. Two block rams were used for the delay lines, slice flip-flops were used to form the 3×3 neighborhood, and the complete $G = |G_x| + |G_y|$ result was computed in the *Convolve block*. No multipliers were required since the kernel values are powers of 2. The required additions and absolute values were done combinationally. The implementation required 113 slices and runs at 302MHz. The design description required approximately 400 lines of VHDL code.

3 Implementing Sobel on the MPPA

The initial MPPA implementation was simple and consisted of only two processors. One processor was used to implement a FIFO, the other implemented the Sobel kernel. The FIFO processor takes an image stream as input and uses line delays to align rows of pixels so that they can be input to the processor that performs the convolution, just as is done for the VHDL implementation. There is a slight difference in the implementation of the FIFO. Instead of using three delays to form the three lines of the neighborhood, the three lines of image data are combined to form a single data stream by successively transmitting 1 pixel value from each row over one channel. The processor running Sobel reads the three values sequentially from the one channel. This has no impact on performance because the Ambric processor has to execute 3 sequential instructions to read and synchronize the input values whether the values are transmitted through one channel or three.

Part of the inner-loop code for the Sobel processor is shown below. The Pn values contain the individual pixel values that were read in from the channels. Px is the value of the convolution with the x-mask; similarly, Py is the value of the convolution with the y-mask. The comparisons against zero implement absolute-value computations. Using this program, each pixel is processed in approximately 44 clock periods, resulting in a frame rate of 26 FPS (512 x 512).

Program 3.1 Java Code for Sobel

```
P1=P2;  P2=P3; P3=in.readInt();
P4=P5;  P5=P6; P6=in.readInt();
P7=P8;  P8=P9; P9=in.readInt();
Px = (P3+P6*2+P9)-(P1+P4*2+P7);
if(Px<0)
  Px = -Px;
Py = (P1+P2*2+2+P3)-(P7+P8*2+2+P9);
if(Py<0)
  Py = -Py;
output = Px+Py;
out.writeInt(output);
```

The one-processor implementation was then evolved into a fine-grained, multi-processor version that achieves a high-level of concurrency. This highly concurrent version uses 18 processors. Each processor performs 1-2 reads, a single arithmetic operation (add, 2x multiply, absolute value, etc) and 1-2 writes. The programs for each of these processors was programmed to be as short as possible so that the pixel initiation interval could be as short as possible. The final version required a maximum of 7 clock cycles per pixel (a

6.3x reduction from the simple one-processor version).

This highly pipelined version is not particularly efficient but that is to be expected as the MPPA is a processor-based array. As an example of one of the fine-grained programs, see Program 3.2. In this code example, processors spend most of their time reading and writing channels — 2 reads (fan-in), a single operation such as an add, followed by 1 write. Thus, processors are only computing values approx-

Program 3.2 Java Code For Simple Pipeline Stage

```
int a = in1.readInt();
int b = in2.readInt();
int res = a+b;
out1.writeInt(res);
```

imately 25% of the time, a direct result of the fine-grained implementation strategy used here.

Compared with the FPGA, the Ambric implementation runs approximately 1/7 as fast. Although this may seem slow in comparison, this is a strong showing for a processor-based array on such a fine-grained computation. We didn't expect the MPPA to perform this well when used in this way. At the rate of 7 clock cycles per pixel, the MPPA can achieve a frame rate of 164 FPS for 512 x 512 images. Alternatively, the MPPA implementation can process high-definition images at 30 FPS. This implementation uses 18 processors out of a total of 336. Note that 9 processors are used to implement buffers to align pixels into rows that are passed to the processors that implement Sobel. All images are composed of 8-bit pixels. Table 1 summarizes the differences between the FPGA and MPAA implementations of the Sobel algorithm.

4 Ambric MPPA Versus FPGA

As devices, the MPPA and FPGA differ in the following fundamental ways:

- **Granularity in Space:** Spatially, the MPPA is more coarse-grained than an FPGA. Whereas the computational element of the FPGA is a combinational 5- or 6-input Look Up Table (LUT) with a 1-bit output, the computational element of the MPPA is a general-purpose, 32-bit, sequential processor. Minimum-width wires are 1-bit wide on FPGAs but are 32-bits wide on the MPPA.

- **Granularity in Time:** Temporally, the MPPA is also more coarse-grained than an FPGA. Temporal granularity refers to how many operations can be performed during some time unit, e.g., a clock-tick. Due to the processor-centric nature of the MPPA, the number of operations that can be performed in a single clock-tick is fixed. On the FPGA, the number of operations that can be performed in a clock-tick is completely variable as long as timing is met. For example, on the MPPA it is possible to perform either an add-operation or a bit-wise or-operation in a single clock-tick, but not both. On the FPGA, you would typically be able to do both operations - packing two operations in a single clock cycle - as long as timing is met. Being temporally fine-grained, the FPGA can more finely distribute operations across clocks to fill each clock period with as many operations as possible. Fine-grained "operator packing" of this sort is not possible with instruction-set processors.

- **Communication:** The MPPA provides only one means of communication: point-to-point, blocking, 32-bit channels that can be used to transmit/receive data from processors, I/O and memory. FPGAs are far more flexible and just about any conceivable communication scheme can be implemented with wires, LUTs and FFs, at the cost of design time.

- **Synchronization:** The MPPA provides a single implicit synchronization scheme that is built on top of the communication channels. Each channel is essentially a 2-element FIFO. Processors stall when reading an empty channel or when writing to a full channel. FPGAs support just about any synchronization scheme at the cost of design time.

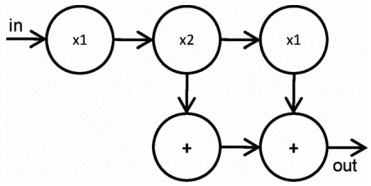
4.1 A Simple Example

To understand how MPPA differences impact performance, relative to an FPGA, consider the simplified computation depicted in Figure 4. The computation in the figure represents part of the Sobel calculation, essentially a subset of the previously described Ambric implementation. It consists of the computation of the y-gradient for the top-row of the 3x3 neighborhood. The general dataflow is shown in part (a) of the figure. Pixel values arrive on the left-hand side. The top row of 3 pixels is formed as the pixels shift across the 3 modules along the top (x1, x2, x1) where each module multiplies its captive pixel value by 1 or 2. (Note that the middle (x2) module passes the non-modified pixel value to its module on the right but transmits the multiplied value to the adder.) Next, these multiplied pixel values are passed to 2 adder modules to be summed together. The final value is output from the final adder at the right-hand side of the figure.

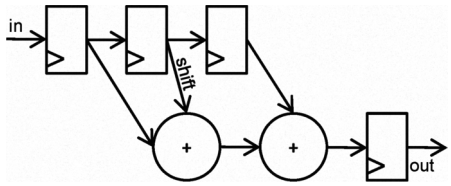
Part (b) of the figure depicts a typical FPGA implementation of this computation. The three modules have been replaced with 3 registers (pixel-bit-width). Similar to the

	Resource Usage	Clocks per pixel	Clock Rate	Frame Rate for 512 x 512 Images
MPPA	18 processors	7	300 MHz	164
FPGA	113 Slices	1	302 MHz	1148

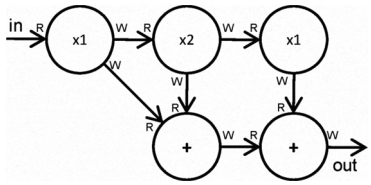
Table 1. FPGA and MPPA Comparison



(a) General Dataflow



(b) FPGA Implementation



(c) MPPA Implementation

Figure 4. Convolution Fragment

computation shown in part (a), the output of two of the registers fans out to two places: the next register and the adder. The x2 is typically implemented in routing as a shift in the FPGA. Note the addition of a pipeline register on the output of the final adder, also typical for an FPGA implementation. Assume that all registers are clocked by the same global clock.

Part (c) of the figure depicts an MPPA implementation of the same kind used in this study. Each of the circles represents a single processor in the MPPA. Three of the processors perform a simple operation, either an addition or a multiplication by 2. Two of the processors are used simply to stage data (those that multiply by 1). Each processor is annotated with text that represents the I/O operation it performs in addition to is computation. 'R' stands for "read from channel" while 'W' stands for "write to channel". For example, the processor that implements the first addition (the left-most adder) must perform 2 reads, one from each input channel, add the received values together, and write the result to its output channel - that processor performs a total of 4 operations.

4.2 Differences and Performance Impact

The following discussion will focus on how differences between the MPPA and FPGA architectures impact performance, relative to the Sobel algorithm.

- **Synchronization:** In the FPGA example, all registers are synchronized to a single clock. When a clock-edge arrives, for example, the additions can commence operation immediately. With Ambric, each computational element must explicitly synchronize each value that is read from a channel. Synchronization for every input leads to a much simpler programming model, but may reduce performance when data can be carefully staged so that processors never stall as was done in this example. Ambric's synchronization approach is costly for fine-grained approaches because they tend to have more I/O and less computation. For this example, processors are computing results about 25% of the time with the remainder used to synchronize and perform I/O. The FPGA uses only one synchronization event - the clock edge for both reads for this example. This works out because the designer manually staged the data so no stalls would ever be necessary. However, for less fine-grained approaches, Ambric's synchronization strategy should work well because I/O instructions will represent a smaller percentage of the computation. In these cases, the costs for explicit synchronization as performed by Ambric will be less, and possibly neglible.

- **Communication:** The main difference between an FPGA and the MPPA, relative to communication, is fan-out/fan-in. Ambric implements communication solely as point-to-point channels. Fan-out must be implemented in time. For example, if a processor wants to send data to n destinations, it must write that data n times to 1 or more channels (this also enables synchronization). FPGAs can implement fan-out in space or time. As such, an FPGA can broadcast a value to multiple destinations in a single clock-tick. However, this flexibility comes at a heavy cost: the need to achieve

timing closure. Similar arguments apply when a processor must read multiple values from various sources.

- **Granularity:** Spatial granularity is mostly related to area-efficiency. The MPPA should be quite area-efficient for wider operations, e.g., 32-bit additions. Temporal granularity, on the other hand, directly relates to performance. For example, on the MPPA, you can only perform a limited, fixed number of computations in a clock-tick. Most of the time, the processors only perform one operation per clock tick though the Java compiler often combined the addition and channel-write into one instruction. However, because the number of operations per clock-tick was predetermined when the device was designed, timing closure is not a concern. This is a big advantage. All operations always work at the maximum clock rate. In contrast, FPGAs can pack multiple computations per clock-tick, as long as timing is met. This is a mixed blessing; designers can make good use of the entire clock-period but achieving timing closure is perhaps the most onerous burden associated with FPGA design.

5 Summary

This paper has described an experiment directed at understanding the relative advantages of FPGAs and MPPAs for a simple, fine-grained image convolution. In the end, the FPGA outperformed the Ambric device by about 7x because the FPGA was better suited to exploit the parallelism exhibited by the Sobel edge-detection algorithm. In particular, fine-grained applications like those found in image processing tend to perform as much (or more) I/O as they do computation. This puts the Ambric device at a disadvantage because it is a processor-based array and processors are simply better suited for applications that require more compute and less I/O.

In the end, most of the FPGA advantage came down to two things:

- the ability to simultaneously transmit data to many destinations and to receive data from many sources in a single clock-tick.

- the ability to pack multiple arithmetic/logic/IO operations in a single clock-tick.

Combining these two features makes it possible to implement a read-modify-write with multiple sources and destinations in a single clock-tick. This is what made it relatively easy to achieve a pixel-per-clock throughput rate for the FPGA.

6 Conclusions and Future Work

Though no match for the raw performance of the FPGA, the Ambric device provided well over real-time performance (164 FPS) for 512x512 images. It should also be noted that other, more coarse-grained approaches to the Sobel algorithm could have been tried and may have achieved better efficiency and throughput, but the scope of this paper was limited to an evaluation of the fine-grained abilities of the Ambric device. In addition, coarser-grained parallel approaches such as those that process multiple subtiles of the image in parallel can be used with the FPGA as well as the MPPA. As such, we anticipate that due to its ability to exploit very fine-grained parallelism, the FPGA would still outperform the MPPA for this algorithm even if coarser-grained parallelism was exploited. However, it is likely that coarser-grained approaches would be more space-efficient for Ambric than the strictly fined-grained approaches that were used on the MPPA in this paper.

In many ways, this represents a worst-case for the Ambric MPPA. The playing field should improve significantly for the MPPA when it is used to implement algorithms that contain more compute than I/O and where the arithmetic operations are wider, e.g., 32-bits or more. The FPGA's ability to simultaneously transmit/receive data from/to multiple sources becomes far less important when it occurs less frequently in the application. In addition, the FPGA's ability to insert multiple arithmetic operations per clock-tick becomes less of an advantage once those operations become wider and consume most of the clock cycle of the FPGA.

For future work, the authors plan to evaluate the applicability of the MPPA with coarse-grained applications such beam-forming, etc.

References

[1] M. Butts. Synchronization through Communication in a Massively Parallel Processor Array. *IEEE Micro*, 27(5):32–40, 2007.

[2] M. Butts, A. Jones, and P. Wasson. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '08)*, pages 55–64, April 2008.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008. General-Purpose Processing using Graphics Processing Units.

[4] B. Hutchings, B. Nelson, S. West, and R. Curtis. Optical Flow on the Ambric Massively Parallel Processor Array (MPPA). In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '09)*, page to appear, April 2009.