Optical Flow on the Ambric Massively Parallel Processor Array (MPPA)

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

The Ambric Massively Parallel Processor Array (MPPA) is a device that contains 336 32-bit RISC processors and is appropriate for embedded systems due to its relatively small physical and power footprint. Optical flow is a computationally-demanding and highly parallelizeable image-processing algorithm with applications in embedded systems such as robotics and autonomous vehicles. An optical flow algorithm is implemented on the Ambric device and is shown to achieve near FPGA performance at similar levels of power consumption while requiring many fewer lines of code (Java) than its FPGA counterpart (VHDL).

1 Introduction

Optical flow refers to the apparent motion of brightness patterns in a scene. In many cases, optical flow is computed by comparing changing brightness patterns in a series of image frames obtained from a video camera. Optical flow has applications in autonomous systems where it can be used to extract important features from the environment to serve as navigational cues and to serve as a guide for motion. For example, an autonomous aircraft could use optical flow to aid in hovering by controlling the movement of the aircraft so that optical flow is close to 0 in all directions. In autonomous applications such as this, it is desirable that the hardware dedicated to computing optical flow is small, lightweight, and high-performance, as optical flow is computationally demanding and real-time performance is necessary for controlling movement of autonomous systems.

Optical flow, like so many other image-processing algorithms exhibits a great-deal of parallelism and high-performance solutions are achievable by technologies that can exploit that parallelism. Some of the highest-performance implementations of optical flow have been achieved with FPGAs and GPGPUs [3, 5].

This paper describes an implementation of optical flow on the Ambric Massively Parallel Processor Array (MPAA). The MPAA contains over 300 programmable processors on a single die. These processors are general purpose and are programmed using assembly code or Java, and communicate over synchronized communication channels. The Ambric MPAA is appealing because it is relatively low power (12 watts or less), is quite flexible because it contains 300 processors and is relatively easy to program. The MPAA is interesting because it combines structural design with general-purpose programming. Programming the Ambric device ”feels” akin to hardware design, because of its flexibility and structural organization, only much easier, because you write sequential programs for each of the processors instead of low-level VHDL code.

This paper will briefly discuss the Ambric device, the general design strategy for capturing user designs and the implementation of optical flow on the Ambric MPAA. It will also compare the Ambric implementation of optical flow against an implementation on an FPGA in terms of power, performance, and code size.

2 Explanation of Optical Flow

The purpose of optical flow is to compute a velocity vector for each pixel in an image, that vector indicating the pixel's apparent motion in the x and y directions over the past few image frames. Optical flow calculations depend on what is called the “brightness constancy constraint”, which makes the assumption that all changes in the brightness patterns in an image sequence can be attributed to motion (either of the camera or of objects within the field of view) rather than ambient lighting changes.

Many optical flow algorithms have been developed over the past two decades. Recent work by Wei *et al* has described a series of optical flow formulations well suited for real-time computation in hardware [5, 7, 6, 3], providing for

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

the possibility of computing optical flow in real-time for autonomous vehicle navigation and obstacle avoidance. The full derivation of the specific optical flow algorithm used in this work is provided in [6] — the description given below is more intended to help the reader understand the flow of the computation and its computational complexity, in order to better understand the implementations described in succeeding sections of this paper.

The algorithm implemented here combines 3D tensors and a ridge regression estimator to produce $[V_x, V_y]$, the final optical flow velocity vector for each pixel in an input image sequence. The input to the calculation is a 3-dimensional volume of data where x and y are the spatial components and t is the temporal component. The computation is a multi-stage pipelined computation as shown in Figure 1 (which was automatically generated from the Ambric design files described later). The first stage (Gradient) performs 1D convolutions in the x, y, and t dimensions using five-element masks to produce gradient (derivative) image sequences called gx, gy, and gt. The second stage (3D Smoothing) then performs smoothing convolutions on each of them to produce gxs, gys, and gts. The 3D convolution is performed as a 3×1 convolution (in time) followed by a 5×1 (in x) followed by a 1×5 (in y).

The third processing stage (Outer Product) produces six new matrices (xx, yy, tt, xy, yt, xt) by performing point-by-point multiplications between elements of the previously computed and smoothed gxs, gyx, and gtx images. A 2D smoothing step is then performed in each of the six succeeding $smooth\_*$ blocks (each does a 3×1 followed by a 1×3 convolution). The result is the creation of the xxs, yys, tts, xys, yts, and xts images.

The "Velocity" block in the figure next produces the velocity field. This is done in multiple steps. First, the velocity field is computed for each pixel as:

$$V_x = \frac{(xxs \times yts - xys \times xts)}{(xxs \times yys - xys^2)}$$
$$V_y = \frac{(yys \times xts - xys \times yts)}{(xxs \times yys - xys^2)} \quad (1)$$

The problem with this computation is that if the smoothed gradients (gxs and gys produced earlier in the pipeline) are nearly linearly dependent (one is nearly a scaled version of the other), small amounts of noise in the image sequence will result in large errors in the velocity field just computed (see [6] for a detailed derivation of this and its solution using a ridge estimator). This can happen in areas of low texture in the image sequence. Further, because system resources are less abundant in the real-time systems described here, the range of the calculation may be constrained (for example, the calculation in both the Ambric and FPGA implementations described later are fixed-point calculations). The result of this reduced range is a higher probability of collinearity causing a problem.

The solution described in [6] is that for each velocity estimate produced, the common denominator from (1) is compared to a threshold. If it does not exceed the threshold then the following is done: (a) a scalar weighting factor k is computed, (b) k is added to xxs and yys, (c) and the velocity calculation in (1) is repeated. The calculation of k is given as:

$$k = \alpha \times (tts - 2xts \times V_x + xxs \times V_x^2 - 2yts \times V_y + 2xys \times V_x \times V_y + yys \times V_y^2) \quad (2)$$

where α is a constant.

The results of the velocity calculation, V_x and V_y, are then spatially smoothed in the final pipeline stage with 7×1 and 1×7 kernels to produce the final velocity fields.

Thus, as can be seen, the calculation is a feed-forward processing chain consisting mainly of convolutions in x, y, and t used for calculating gradients and performing smoothing operations. These are combined with outer product calculations and a velocity calculation to produce the result. Finally, note that the test of k against the threshold and the subsequent re-computation of the final velocity field as described above is the only data-dependent computation in the pipeline.

3 An Introduction to Ambric

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

The AM2045 is internally organized into a 2D array of *bric* modules. Each bric contains 8 CPUs (each with its own internal RAM), 2 memory objects (each organized as 4 independent RAM banks), and local interconnect between them. Level 2 communication channels provide direct connections to neighboring brics. Level 3 interconnect is for long-distance communications and consists of a chip-wide 2D circuit-switched interconnect of channels.

The AM2045 core is connected to the following external interfaces:

- Two 32-bit DDR2-400 SDRAM interfaces,
- 4-lane PCI Express that is used for chip configuration and data transport,
- 128 1-bit general-purpose parallel I/O ports,
- serial flash, microprocessor, and JTAG interfaces.

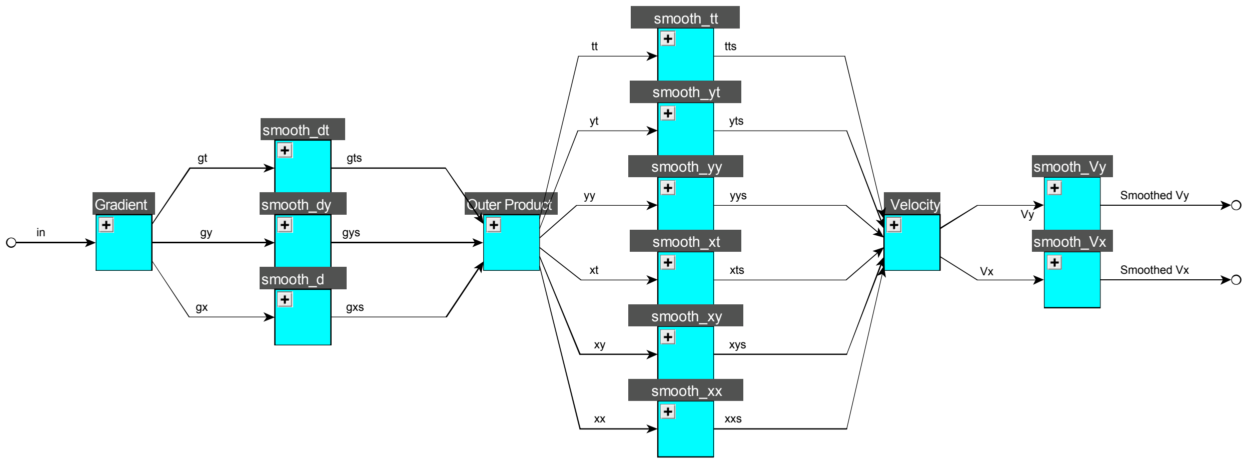

Figure 1. Ambric Optical Flow Implementation

The AM2045 is mounted on the Ambric IDB board that provides 256 MB of DDR RAM and the other above-mentioned I/O interfaces. For this application, the on-board DDR was used to create deep FIFOs that were used to buffer images. Additional details of the AM2045 can be found in [1, 2].

4 Overview of Ambric Programming Model

There are two basic primitive objects in the Ambric programming model:processors and memories. Processors are 32-bit RISC machines and come in two varieties: SRD and SR. 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 memory objects. SR processors are lighter weight and contain only 1 ALU and are used for address generation and for accessing memory objects. They contain a dedicated 128-word memory for programs and data but do not have direct connections to memory objects. Note that processors do not share memory but communicate only via point-to-point channels.

Memory objects consist of four independent single-port RAM banks, each containing 256 words. These memory objects 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 through 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 (valid bit). Reads from an empty channel typically cause a processor stall as do writes to full channels. Nearest-neighbor objects communicate directly over non-shared channels for highest bandwidth. Objects that are further away also communicate via channels but these longer channels share physical resources and provide less bandwidth. 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. Channels provide at least two major benefits to the Ambric programmer. First, they relieve the programmer of the onerous task of explicitly synchronizing hundreds of processors - synchronization is completely automatic - contrast this with typical thread-based programming [4]. Second, they encapsulate processors and memories. This makes it much easier to reason about the system's behavior (no side effects in memory) and each object can be treated independently, making it easier to program each individual processor.

Programmers develop applications on the Ambric MPPA by writing small Java programs, one per processor. The programmer is also responsible for providing a "structural" description of their application that assigns programs to processors and defines the source and destinations for channels. This structural description is somewhat akin to a circuit net-list where the wires are channels and the cells are processors.

Initially, the process of writing programs for 300+ processors sounds daunting but turns out to be manageable in practice for the following reasons:

- *Programs are quite small.* Each processor only provides 128-256 words of dedicated memory. Java programs for this application tended to be less than 50 lines of code.

- *Programs are heavily reused.* the number of distinct programs tends to be few in number, relative to the number of processors. In a typical application with lots of exploitable parallelism, high performance is often achieved, for example, by concurrently executing several *identical* operations simultaneously, on different sections of the data stream. Each identical operation uses a distinct processor, but uses the same compiled program code. In other examples, complex data operations can often be created by composing several identical or similar operations. For the optical-flow application, there are only 27 distinct Java programs spread across 126 processors.

- *Synchronization is implicit.* Unlike a typical thread-based program that relies on explicit locking for synchronization, the Ambric MPPA relies on a data-flow model that simply blocks until data is available. This makes programs simpler and smaller.

- *Applications are hierarchically organized.* Applications are developed hierarchically: processors implement primitive operations; these primitive operations are grouped together to form more complex operations, and so forth. Hierarchy simplifies development by hiding detail and making it easier to reuse code and processor organizations.

Taken together, these four things (small programs, reuse, implicit synchronization, hierarchy) make it feasible to implement applications on the Ambric processor. They also provide a design process reminiscent of that used to design hardware. In hardware, for example, functional units are coded in HDL, connected with wires, and grouped hierarchically to create more complex and powerful computations. Analogously, in the Ambric MPPA, processor units are coded using Java, connected using channels, and are grouped hierarchically to create complex objects consisting of multiple processors, channels and programs.

5 Optical Flow on the Ambric MPPA

The previously-described optical flow algorithm was implemented on the Ambric 2045 on the Ambric IDB board. The algorithm occupies about 60% of the capacity of the Ambric device. General resource utilization is shown in Table 1 while Table 2 details the resource usage of the program blocks shown in Figure 1, from left to right, and documents the number of files for both the Java and aStruct source files, the number of lines of code in each file, and the number of processor and memory units. Note that the shift-saturate module is not a top-level module but is utility code that is used extensively (30 times) throughout the application.

Resource Type	**Used**	**Free**	**Total**	**% Used**
SRD processor	128	40	168	63.10
SR Processor	62	104	166	31.33
Ram Object	162	174	336	47.92

Table 1. Overall AM2045 Resource Usage for Optical Flow

5.1 Example Block: The Derivative Calculation

A complete discussion of all of the blocks that comprise the optical-flow application is beyond the scope of this paper. A decomposition of part of the $derivative$ block, starting at the top level and proceeding through the bottom of the hierarchy will be detailed below to give the reader a glimpse of the details of the application, as implemented on Ambric.

Consider the first stage of processing in Figure 1 — the Gradient block, known as the *derivative* block in the Ambric design, shown at the far left of the figure. It consists of six java files and six astruct files, which are 301 lines of code and 109 lines respectively, a total of 410 lines of code.

The purpose of the derivative block is to accept a sequence of images and then perform three convolutions. One convolution is temporal (the dt calculation) and is computed across five images. The other two convolutions are spatial in one dimension (dx and dy), operate on only one image at a time and use five-element masks.

Dropping down one level in the hierarchy, the design of the derivative block is given in Figure 2, that shows that it is composed of several hierarchical and primitive blocks. In the picture, the small square blocks are leaf nodes or primitive functions consisting of a single processor — their functionality is provided by Java or assembly code. The larger blocks are hierarchically composed of lower level blocks and thus contain multiple processors.

In Figure 2, starting at the far left, raw image data enters the $split$ block that creates data streams for the dt, dx, and dy blocks that compute the actual derivatives. The three different derivative calculations compute results using differing numbers of images. The dx and dy perform spatial convolutions and only require one image at a time. The dt block performs a temporal convolution across five images.

Delay blocks dx_delay and dy_delay are hierarchical blocks used to synchronize data for the dx and dy blocks. The delay modules implement the control protocol required by the DDR controller and provide FIFO behavior. These

Block Name	File Count (Java, aStruct)	Line Count (Java, aStruct)	Processor Units	Ram Units
Derivative	4, 2	301, 109	14	6
Smooth	4, 3	200, 130	39	30
Outer Product	2, 1	41, 110	14	6
Smooth	2,1	84, 90	30	36
Velocity	9, 5	358, 508	53	33
Smooth	4, 1	230, 131	8	20
Shift-Saturate	2, 3	85, 65	-	-
Top	0, 1	0, 69	-	-
Total	27, 17	1324, 1212	158	131

Table 2. Detailed Resource Usage for Top-Level Blocks

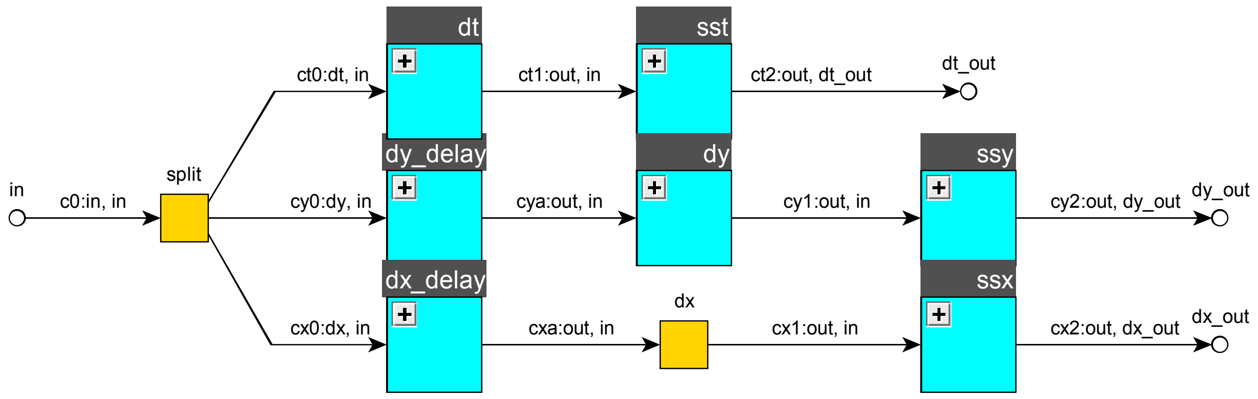

Figure 2. Design of the Derivative (Gradient) Block

blocks initially fill their respective off-chip memory areas with image data and then read data one element at a time, overwriting the outbound data with incoming data.

The dx block is a primitive block that performs a simple spatial convolution with a single processor. The dy block is a hierarchical block; its additional complexity is due to additional reordering and buffering required to compute the spatial derivative in the y-direction (image data arrive in x order). The dt block is also hierarchical and contains larger DDR-based (off-chip) buffers for buffering and reordering data from the incoming five images that it uses to compute the temporal derivative. Finally, the ss_* blocks are scaling blocks required since the computation is being done in fixed point arithmetic.

Dropping down one more level in the hierarchy, Figure 3 shows the the contents of the dt block shown near the top-left of Figure 2. It consists of a primitive $control$ block that feeds the DDR buffer (another hierarchical block) which, in turn, feeds the dt block that actually performs the temporal computation. The result is passed to the $fifo\_dt\_out$ block that provides the final output for the composite dt block. This output feeds the sst block up a level in the hierarchy, shown in Figure 2.

Finally, dropping to a leaf level, Program 5.1 shows the yellow primitive dt block that implements the temporal convolution mask that consists solely of the Java code used to compute the convolution.

5.2 Another Example: The Split Block

For a more complex coding example, consider the $split$ block, also in the $derivative$ block, another primitive block implemented on a single processor - its implementation consists of the Java code shown in Program 5.2. After sending the first two frames' pixels to the dt block only, it then sends portions of all remaining frames to dx, dy, and dt.

The code describing the elimination of the first two frames has been removed for brevity in this example. The control logic in the program determines where each pixel read from the input stream is forwarded to: (1) the first and last two rows of pixels in the image are only forwarded to dy, and (2) for the middle rows, the first two pixels are sent to dx, the middle pixels to dx, dy, and dt, and the last two pixels in the row are sent to dx.

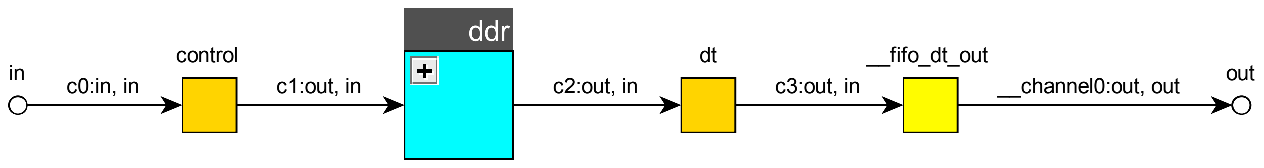

Figure 3. Design of the Composite dt Block

Program 5.1 Code for the *dt* Convolution

```
package derivative;
import ajava.io.InputStream;
public class Dt Convolution
{
public void run(InputStream<Integer> in,
                OutputStream<Integer> out)
  {
    // Data comes from ddr as serialized stream
    int p0=in.readInt();
    int p1=in.readInt();
    int p3=in.readInt();
    int p4=in.readInt();
    // Mask is: [-1 8 0 -8 1]
    out.writeInt(-p0+p1*8-p3*8+p4);
  }
}
```

5.3 Structural Descriptions in Ambric Designs

The structural programming model used by Ambric consists of two major pieces: Java or assembly code that is bound to individual processors and a structural file (astruct) that describes how the processors communicate over channels. The format also supports parameterized conditional and iterative instantiation for additional flexibility. The astruct file for the composite dt block shown in Figure 3 is illustrated in Program 5.3. An astruct file flows as follows: declare modules, instance them, and connect them together over channels. For example, in Program 5.3 the DDR2 object is imported, declared and given several parameters. Also, channel c0 connects the input of the block to the control object input, while channel c1 connects the output of the control block to the input of the ddr object, as described in Program 5.3 and depicted in 3.

5.4 Ambric Performance

At press time, the optical-flow design achieved approximately 37 frames per second (FPS) on an image stream

Program 5.2 Code for the *split* Block

```
public void run(InputStream<Integer> in,
                OutputStream<Integer> dx,
                OutputStream<Integer> dy,
                OutputStream<Integer> dt)
{
  int w=this.width;
  int h=this.height;

  while(true)
  {
  // First two rows go to dy only
  for(int i=0;i<2;i++)
    for(int j=0,w_4=w-4;j<w_4;j++)
      dy.writeInt(in.readInt());

  // Middle rows
  for(int k=0, middle_rows=h-4;
       k<middle_rows;k++)
  {
    // First two pixels in row go to dx only
    dx.writeInt(in.readInt();
    dx.writeInt(in.readInt());

    // Middle pixels in row go to all 3
    for(int l=0, mid_col=w-4;l<mid_col;l++)
    {
      int pixel=in.readInt();
      dx.writeInt(pixel);
      dy.writeInt(pixel);
      dt.writeInt(pixel);
    }

    // Last two pixels in row go to dx only
    dx.writeInt(in.readInt());
    dx.writeInt(in.readInt());

  }

  // Last two rows go to dy only
  for(int i=0;i<2;i++)
    for(int j=0,w_4=w-4;j<w_4;j++)
      dy.writeInt(in.readInt());
}
```

Program 5.3 Astruct Code for the Composite dt Block

```
import astruct.io.DDR2;
interface Dt_
{
 inbound in;
 outbound out;
}
binding jDt implements Dt_
{
 implementation "Dt.java";
}
interface Dt
{
 inbound in;
s outbound out;

 property int width;
 property int height;
}
binding aDt_w_fifos implements Dt
{
 DDR2 ddr = controller=0,
             port = 0,
             size=width*height*5;//w*h*4*5
 DDR_Control control =width=width,
                        height=height;
 Dt_ dt;

 channel c0 = in, control.in;
 channel c1 = control.out,ddr.in;
 attribute MinimumBuffer(512) on control.out;
 channel c2 = ddr.out, dt.in;
 channel c3 = dt.out, out;
}
```

consisting of 320x240 images. Initially, the design achieved about 25 FPS but achieved the higher figure after we optimized the image buffering schemed that used in the Derivative block. Image data for this experiment were generated on-chip because of limitations in the I/O interfaces[1]. While a detailed comparison against an FPGA implementation is beyond the scope of this paper, Table 3 gives a rough idea of the relative performance of the FPGA and Ambric implementations. The FPGA implementation cited here is the one reported by Wei *et al* [6] and was done on a Virtex-4 FX60 part. As shown, the Ambric implementation is roughly on par with the FPGA implementation, currently achieving over 1/2 the throughput with about 20% lower power consumption. Note that the SDRAM consumes 1.23 watts of the total 8.4 watts consumed by the Ambric device while running the optical-flow application.

5.5 Analyzing Ambric Performance

The Ambric MPPA device contains a facility to statistically measure the performance of each processor as they execute in hardware. Processors are polled for their current state (running, stalled on input, stalled on output). This state is then transferred over the debug network to the host device. This is repeated for each active processor in a round robin fashion. A meaningful measurement can be obtained after several seconds of execution. This is essentially a rough form of profiling and can be used to locate bottlenecks and to optimize the design. We used this profiling tool extensively to find bottlenecks and optimize the design.

The performance of the optical flow implementation was measured using the statistical measurement facility and found to be active about 18% of the time. Most of the time, the application is stalled in input and this is true for all of the modules listed in Table 4. As shown in the figure, all blocks are stalled, waiting for input, 70% or more of the time. These stalls are most likely due to remaining inefficiencies in the FIFO designs that we developed in the Derivative block to buffer images for the temporal convolutions. We estimate that nearly another factor of two in performance may be available when this part of the design is fully optimized.

6 Conclusion

In conclusion, the Ambric implementation achieves real-time performance for the optical flow algorithm with a physical and power footprint appropriate for embedding in autonomous systems such as unmanned vehicles, robots, etc. It competes well against a Virtex-4 FX60 FPGA implementation in both power and throughput. The Ambric

[1]Sadly, Ambric ceased operations during this project. We were unable to resolve this problem because they were unable to offer support.

Implementation	Image Throughput (320x240)	Power	Code Size in lines
Ambric IDB	37	8.4 watts	2,536
Helios FPGA	60	10 watts	17,000

Table 3. Overall AM2045 Resource Usage for Optical Flow

Module	Running	Stalled on Input	Stalled on Output	Busiest
Derivative	19.67%	74.88%	4.46%	DDR Control, 40% (stalled on out 50%)
Smooth Derivative	20%	80%	0%	Conv_x, 45%
Outer Product	16%	80%	4%	Op2 40.74%
Smooth Outer Product	23%	77%	0%	Conv_x 38%
Velocity	17.75%	68.54%	12.96%	V2.Vel_den 61.82%
Smooth Velocity	11%	88%	0%	Conv_x, 24.53%
Top	18.6%	73.3%	5.7%	V2.Vel_den, 61.82%

Table 4. Profile of Execution

implementation also compares very favorably against the FPGA in terms of code size: 2,500 lines of Java and astruct for Ambric and 17,000 lines of VHDL code for the FPGA implementation. Although relative productivity for the Ambric device and programming model remains to be studied, the dramatic difference in code size strongly hints that the Ambric device may provide substantial productivity gains and yet be able to provide high performance for embedded environments.

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] J. Chase, B. Nelson, J. Bodily, W. Z., and L. D.J. Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '08)*, pages 173–182, April 2008.

[4] E. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, 2006.

[5] Z. Wei, D. Lee, and B. Nelson. FPGA-based Real-time Optical Flow Algorithm Design and Implementation. *Journal of Multimedia*, 2(5):38–45, 2007.

[6] Z. Wei, D. Lee, B. Nelson, and J. Archibald. Real-time Accurate Optical Flow-based Motion Sensor. In *IEEE International Conference on Pattern Recognition (ICPR)*, December 2008.

[7] Z. Wei, D. Lee, B. Nelson, J. Archibald, and B. Edwards. FPGA-Based Embedded Motion Estimation Sensor. *International Journal of Reconfigurable Computing*, 2008, July 2008.